



The background of the book cover features a vertical striped pattern. The stripes are primarily blue, with varying shades creating a sense of depth. Interspersed among the blue stripes are horizontal bands of bright green, which appear to be slightly darker than the blue. The overall effect is reminiscent of a modern architectural facade or a stylized landscape.

# DOMAIN-DRIVEN **DESIGN** DESTILADO

VAUGHN VERNON



# DOMAIN-DRIVEN DESIGN DESTILADO

---

VAUGHN VERNON

*Traducción*

*Carlos Buenosvinos*

*Leonardo Cano*

*Nicole Andrade*

*Revisión técnica*

*Yuji Kiriki Rodríguez*

*Nicole y Tristan  
¡Lo logramos de nuevo!*

# Contenido

Prefacio . . . . .	xii
A quién va dirigido este libro . . . . .	xiii
Convenciones . . . . .	xiv
Agradecimientos . . . . .	xv
Acerca del autor . . . . .	xvii
<b>Capítulo 1 Por qué usar DDD. . . . .</b>	<b>1</b>
¿DDD será doloroso? . . . . .	2
Diseños buenos, malos y efectivos . . . . .	3
Diseño estratégico . . . . .	8
Diseño táctico . . . . .	9
El proceso de aprender y destilar conocimiento . . . . .	10
¡Hora de comenzar! . . . . .	10
<b>Capítulo 2 Diseño estratégico con <i>Bounded Contexts</i> y <i>Ubiquitous Language</i> . . . . .</b>	<b>11</b>
Expertos del dominio e impulsores de negocios . . . . .	17
Caso de estudio . . . . .	21
Diseño estratégico fundamental requerido . . . . .	24
Beneficios al hacer pruebas. . . . .	25
Desafiar y unificar . . . . .	29
Cómo desarrollar un <i>Ubiquitous Language</i> . . . . .	35
Poner en práctica los escenarios . . . . .	39
¿Qué ocurre a largo plazo? . . . . .	41
Arquitectura . . . . .	42
Resumen . . . . .	44
<b>Capítulo 3 Diseño estratégico con subdominios . . . . .</b>	<b>45</b>
¿Qué es un subdominio? . . . . .	46
Tipos de subdominios . . . . .	46

Cómo confrontar la complejidad. . . . .	47
Resumen . . . . .	50
<b>Capítulo 4 Diseño estratégico con <i>Context Mapping</i>. . . . .</b>	<b>51</b>
Tipos de mapeo . . . . .	54
Alianza . . . . .	54
Kernel compartido . . . . .	55
Relación cliente-proveedor . . . . .	55
Relación conformista . . . . .	56
Capa anticorrupción ( <i>Anticorruption layer</i> ) . . . . .	56
Servicio abierto . . . . .	57
Idioma publicado ( <i>Published Language</i> ) . . . . .	58
Caminos separados ( <i>Separate Ways</i> ) . . . . .	58
<i>Big Ball of Mud</i> . . . . .	59
Cómo hacer uso apropiado del <i>Context Mapping</i> . . . . .	60
RPC con SOAP . . . . .	62
RESTful HTTP . . . . .	63
Mensajería . . . . .	65
Ejemplo de <i>Context Mapping</i> . . . . .	70
Resumen . . . . .	74
<b>Capítulo 5 Diseño táctico con agregados . . . . .</b>	<b>75</b>
¿Por qué se utilizan? . . . . .	76
Reglas básicas de los agregados . . . . .	80
Regla 1: proteger las invariantes del negocio . . . . .	
dentro de límites del agregado . . . . .	81
Regla 2: diseñar agregados simples . . . . .	82
Regla 3: referenciar otros <i>agregados</i> usando . . . . .	
únicamente su identidad . . . . .	84
Regla 4: actualizar otros agregados usando <i>Eventual Consistency</i> . . . . .	85
Modelamiento de agregados . . . . .	88

Elegir las abstracciones con sumo cuidado. . . . .	93
Dimensionar agregados correctamente . . . . .	95
Unidades verificables . . . . .	98
Resumen . . . . .	98
<b>Capítulo 6 Diseño táctico con <i>Domain Events</i> . . . . .</b>	<b>101</b>
Diseñar, implementar y usar <i>Domain Events</i> . . . . .	102
<i>Event Sourcing</i> . . . . .	109
Resumen . . . . .	112
<b>Capítulo 7 Herramientas de gestión y aceleración. . . . .</b>	<b>113</b>
<i>Event Storming</i> . . . . .	114
Otras herramientas . . . . .	126
Cómo gestionar DDD en un proyecto Agile . . . . .	127
Primero lo primero . . . . .	128
Utilizar análisis DOFA. . . . .	129
<i>Modeling Spikes</i> y <i>Modeling Debt</i> . . . . .	130
Identificar tareas y estimar esfuerzos. . . . .	132
<i>Timeboxed Modeling</i> (modelo acotado en tiempo). . . . .	134
Cómo implementar . . . . .	135
Cómo interactuar con los expertos del dominio . . . . .	136
Resumen . . . . .	138
Referencias. . . . .	139
Lista de términos traducidos . . . . .	141
Índice analítico. . . . .	143



# Prefacio

¿Por qué será que construir maquetas es una actividad tan entretenida y gratificante? Desde pequeño, me han encantado las maquetas. En aquel entonces, pasaba horas construyendo réplicas de vehículos y aviones. No estoy seguro si LEGO® existía por aquella época. Sea como sea, LEGO ha sido una parte importante de la vida de mi hijo desde que era muy pequeño. Es fascinante diseñar y construir modelos con esas piezas tan pequeñas. Me sorprende lo fácil que resulta crear modelos sencillos y hacerlos cada vez más grandes. El límite es la imaginación.

Supongo que todos tenemos algún recuerdo de chicos jugando y haciendo maquetas con modelos.

Los modelos están presentes en muchas situaciones cotidianas. Si uno disfruta los juegos de mesa, lo hace usando un modelo. Por ejemplo, un modelo de bienes raíces o sobre islas y sobrevivientes, o quizás un modelo sobre ocupar terrenos y construir edificaciones. De forma similar, los videojuegos también son modelos. Quizás representan un mundo maravilloso lleno de personajes fantásticos con misiones increíbles o tal vez un videojuego sobre cartas con poderes mágicos. Usamos modelos todo el tiempo y probablemente de forma tan habitual que no los valoramos como se merecen. Los modelos forman parte de nuestras vidas.

Pero, ¿por qué? Pues porque cada persona tiene una forma distinta de aprender. Existen varias formas de aprendizaje; tres de las más relevantes son: auditiva, visual y táctil. Las personas auditivas aprenden oyendo y escuchando; las visuales lo hacen mediante la lectura y las imágenes, mientras que las táctiles aprenden con el sentido del tacto. Es interesante saber que cada individuo tiene un estilo de aprendizaje claramente predominante sobre los otros dos, hasta tal punto, que una persona puede tener serias dificultades al usar su forma no natural. Por ejemplo, quienes aprenden mediante el tacto recordarán lo que hicieron pero no lo que se les explicó durante el proceso. En lo que a hacer maquetas se refiere, quizás las personas visuales y táctiles tengan ventaja sobre las auditivas, ya que parece que modelar requiere en especial los sentidos de la vista y el tacto. Sin embargo, no siempre es cierto, en particular si un equipo usa la comunicación verbal en su proceso para construir maquetas. En otras palabras, la construcción de modelos ofrece la posibilidad de adaptarse al estilo de aprendizaje de la gran mayoría de las personas.

Con nuestra forma natural de aprender usando modelos, ¿por qué no modelar el software que cada día más impacta en nuestras vidas? De hecho, modelar software parece ser indudablemente una tarea humana. Y modelar software deberíamos. Parece que los humanos deberíamos ser la élite del modelado de software.

Mi propósito es ayudar a descubrir la parte más humana usando algunas de las mejores herramientas disponibles para modelar software. Estas herramientas se agrupan bajo el enfoque de “Domain-Driven Design” o (DDD). Este conjunto de herramientas, o patrones y prácticas siendo más específicos, fue acuñado por primera vez por Eric Evans en su libro *Domain-Driven Design: Tackling Complexity in the Heart of Software [DDD]*. Mi visión es llevar DDD a todos los rincones del mundo. Y para ser más concreto, hacer masivo DDD, que es donde merece estar. DDD es el conjunto de herramientas que toda persona necesita usar para crear los modelos más avanzados de software. Con este libro, el compromiso es hacer del aprendizaje y uso de DDD lo más sencillo posible, así como su difusión entre el mayor número de personas posible.

Para las personas auditivas, DDD ofrece la posibilidad de aprender a través de la comunicación en equipo para construir un modelo basado en el desarrollo de *Ubiquitous Language* (lenguaje ubicuo). Para las visuales y táctiles, el proceso de usar herramientas que DDD provee es muy visual y práctico, ya que el equipo modela de forma táctica y estratégica. Esto es especialmente cierto al dibujar *Context Maps* (mapas de contextos) y modelar procesos de negocios usando *Event Storming* (lluvia de eventos). Por tanto, DDD ayuda a todos los que quieran aprender y lograr cosas grandes mediante la construcción de modelos.

## A quién va dirigido este libro

---

Este libro es para todos aquellos interesados en aprender brevemente los aspectos y las herramientas más importantes de DDD. Con frecuencia son arquitectos y desarrolladores de software quienes aplicarán DDD en sus proyectos. Muy a menudo y rápidamente, los desarrolladores de software descubren lo llamativo de DDD y se ven seducidos por sus potentes herramientas. Y con esto en mente, he preparado el material para que sea comprensible por ejecutivos, expertos de dominio, gerentes, analistas de negocios, arquitectos de información y responsables de calidad. En

realidad, es ilimitado para todos aquellos que trabajen en el sector de las tecnologías de la información (TI) y en investigación y desarrollo (I+D).

Si un consultor está trabajando con un cliente al que se le puede recomendar DDD, es una gran oportunidad para hacerle llegar este libro como una manera de hacer que las partes interesadas se pongan al día rápidamente. Si son desarrolladores, de cualquier nivel, y trabajan en un proyecto con un cliente, y no están familiarizados con DDD, igualmente es el momento de leer el libro. Como mínimo, todas las partes involucradas en el proyecto podrán asimilar el vocabulario y comprender las herramientas principales de DDD. Esto les permitirá compartir e intercambiar conceptos significativos a medida que avanza el proyecto.

Sea cual sea la experiencia y labor del usuario, es importante leer este libro y poner en práctica DDD en un proyecto. Después, volver a leerlo y verificar lo que aprendido de las experiencias para mejorar en el futuro.

## Acerca de la traducción

---

Dado que esta publicación utiliza tecnicismos propios del ámbito de DDD, se decidió utilizar en la mayoría de los casos el criterio editorial de respetar los términos originales de la edición en inglés. Esto obedece a que en la práctica, para ser más precisos, los equipos de proyectos de DDD utilizan esta terminología sin traducirla a otros idiomas. Para ello, se establecieron dos variantes que aplican cuando se presenta por primera vez un término dentro del libro: 1) conservar la expresión en inglés: se identifica mediante letras itálicas con la traducción al español entre paréntesis y letras redondas, por ejemplo, *Ubiquitous Language* (lenguaje ubicuo); 2) mantener la expresión traducida al español: se presenta en letras redondas, junto a la expresión original en inglés entre paréntesis y letras cursivas. Por ejemplo, capa anticorrupción (*Anticorruption Layer*).

Así, quienes utilicen el libro, como material de consulta o como guía indispensable para el desarrollo de proyectos DDD, tendrán la posibilidad de contar con las dos versiones de la terminología. En este sentido, como complemento a lo anterior, el libro incluye el apéndice “Lista de términos traducidos”, que le permitirá al lector consultar los diferentes tecnicismos con sus equivalentes inglés-español.

## Contenido

---

El primer capítulo, “Por qué usar DDD”, explica qué puede hacer DDD por el usuario y su organización. Proporciona una detallada visión sobre qué va a aprender y por qué es importante.

El capítulo 2, “Diseño estratégico con *Bounded Contexts* y *Ubiquitous Language*”, introduce el diseño estratégico con DDD y enseña sus piedras angulares, *Bounded Contexts* y *Ubiquitous Language*. El capítulo 3, “Diseño estratégico con subdominios”, explica el concepto de *subdominios* y cómo usarlos para abordar la complejidad que atañe integrar nuevas aplicaciones con sistemas legados existentes. El capítulo 4, “Diseño estratégico con *Context Mapping*”, enseña las múltiples maneras como diferentes equipos trabajan juntos de forma estratégica y las correspondientes maneras de integrar el software, que se conoce como *Context Mapping*.

El capítulo 5, “Diseño táctico con agregados”, concentra la atención en crear tácticamente modelos usando *agregados*. Otra importante y potente herramienta del modelado táctico son los *Domain Events*, tema del capítulo 6, “Diseño Táctico con *Domain Events*”.

Finalmente, en el capítulo 7, “Herramientas de gestión y aceleración”, el libro resalta algunos trucos para acelerar y herramientas de gestión de proyectos que pueden ayudar a los equipos a establecer y mantener su ritmo de trabajo. Estos dos aspectos con frecuencia no aparecen explicados en otros recursos, aunque son totalmente necesarios para aquellos determinados a implementar DDD.

## Convenciones

---

Existen algunas convenciones para tener en cuenta al leer el libro. Todas las herramientas sobre DDD que utilizo aparecen en *cursiva*. Por ejemplo, conceptos como *Bounded Contexts* y *Domain Events*. Otra convención es que cualquier código de ejemplo se presenta con la fuente Courier a un solo espacio. Acrónimos y abreviaciones del material de referencia, que pueden encontrar en las últimas páginas, aparecen en paréntesis cuadrados en el texto de los capítulos.

Con esto en mente, sin embargo, lo que más se enfatiza en el libro, es el aprendizaje visual a través de muchos diagramas e imágenes que impactan el cerebro. El lector podrá ver que no hay referencias numéricas en los diagramas que no generen ningún tipo de distracción. Todos los diagramas e imágenes van precedidos por textos descriptivos, que introducen ideas muy pertinentes sobre las gráficas a medida que se avanza en su lectura, lo que significa el refuerzo textual a cierta imagen o diagrama como apoyo visual.

## Agradecimientos

Éste es mi tercer libro con la reconocida editorial Addison-Wesley. También la tercera ocasión que trabajo con el editor Chris Guzikowski, y Chris Zahn, editor de desarrollo. Tercera vez igualmente placentera al igual que las otras dos. Gracias de nuevo por publicar mis obras.

Como siempre, un libro no podría materializarse con éxito sin retroalimentación constructiva. En esta ocasión, he contactado a varios profesionales de DDD quienes no sólo enseñan o escriben necesariamente sobre DDD sino que también trabajan en proyectos mientras enseñan a otros el poder de la herramienta. Ellos han sido son cruciales para asegurar que este material, cuidadosamente depurado, comunique lo necesario exactamente y en forma adecuada. Es algo así como, si uno quiere hablar durante 60 minutos, necesita 5 minutos para prepararse, pero si quiere hablar durante cinco minutos, necesita varias horas para ello.

Jérémie Chassaing, Brian Dunlap, Yuji Kiriki, Tom Stockton, Tormod J. Varhaugvik, Daniel Westheide y Philip Windley. ¡Muchas gracias!

# Acerca del autor

Vaughn Vernon es un veterano creador de software y líder de ideas en la simplificación del diseño e implementación del software. Es autor de los exitosos libros *Implementing Domain-Driven Design* y *Reactive Messaging Patterns with the Actor Model*. Vernon ha dictado talleres de cómo implementar DDD por todo el mundo a desarrolladores de software y seminarios frecuentes en la industria. Sus áreas de interés son sistemas distribuidos, mensajería y, en particular, el modelo de actores. Asimismo, se ha especializado en consultoría sobre DDD, así como en la aplicación de DDD mediante Actor Model, en su rol de Chief Architect and Founder de vlingo/PLATFORM <https://vlingo.io>. Para actualización permanente de su obra, consulte el blog en <https://Kalele.io> o siguiendo la cuenta de Twitter @VaughnVernon.

# CAPÍTULO 1

## Por qué usar DDD

Las personas normalmente desean mejorar su experticia e incrementar el éxito en sus proyectos. Están ansiosas por ayudar las empresas a competir a niveles más altos con el software que desarrollan. Por ello, desean implementar un software que no solo modele correctamente las necesidades de su organización, sino que también rinda a gran escala utilizando las arquitecturas de software más avanzadas. Aprender *Domain-Driven Design* (diseño guiado por el dominio) y hacerlo rápidamente, puede ayudar a lograr todo esto y mucho más.

DDD es un conjunto de herramientas que ayudan a diseñar e implementar software que proporciona un gran valor, tanto de forma estratégica como táctica. Una organización no siempre puede ser la mejor en todo, por ello es mejor que elija cuidadosamente en lo que puede sobresalir. Las herramientas de desarrollo estratégico de DDD ayudan a cualquier equipo a tomar las mejores decisiones de diseño de software y cómo integrarlo mejor en el negocio. En esa medida, la organización se beneficiará al máximo con modelos de software que reflejen explícitamente sus principales propuestas de valor. Las herramientas de desarrollo táctico de DDD pueden ayudar a cualquier usuario y a los equipos a diseñar software útil que modele con precisión las operaciones específicas de una empresa. Una organización, entonces, puede beneficiarse de las amplias opciones para desarrollar soluciones en una amplia variedad de infraestructuras, bien sea físicas o en la nube. Con DDD, es posible llevar a cabo los diseños de software más efectivos y las implementaciones necesarias para tener éxito en el escenario empresarial altamente competitivo de hoy en día.

Este libro presenta, mediante un mejorado y destilado DDD, las herramientas de modelado estratégico y táctico. Aborda en forma comprensible las exigencias únicas sobre desarrollo de software y los desafíos que enfrentan los usuarios mientras trabajan para mejorar su experticia en una industria muy dinámica. No siempre es posible dedicar meses a leer un tema como DDD, y aplicarlo lo antes posible.

A partir del exitoso libro *Implementing Domain-Driven Design* [IDDD], también se ha creado y dictado el taller de *Cómo implementar IDDD* en tres días. Por ello, el compromiso de hacer esta obra para traer DDD de forma muy cuidadosa y sintetizada. Todo esto forma parte del trabajo y sobre todo el principal objetivo de llevar DDD a todos los usuarios y equipos de desarrollo de software, es decir, donde se necesita.

## ¿DDD será doloroso?

---



Possiblemente con frecuencia se escucha que DDD es un enfoque complicado para el desarrollo de software. Pero, ¿por qué complicado? La verdad es que no se puede definir así por simple naturaleza. De hecho, es un conjunto de técnicas avanzadas por utilizar en proyectos de software que son complejos. Debido a su potencial y a la cantidad de contenido por aprender, sin soporte experto, poner DDD en práctica por cuenta propia muchas veces puede ser desalentador. Probablemente también se haya encontrado que otros libros sobre DDD tienen varios cientos de páginas y son muy difíciles de consumir y poner en práctica. De hecho, ha sido necesario utilizar muchas palabras para explicar DDD en detalle a

fin de proporcionar una referencia exhaustiva sobre cómo implementar más de una variedad de conceptos y herramientas de DDD. Ese esfuerzo dio lugar a *Implementing Domain-Driven Design* [IDDD]. Esta nueva obra, mucho más condensada, permite que el usuario se familiarice con las partes más importantes de DDD de la manera más rápida y sencilla posible. ¿Por qué? Porque muchas personas se sienten confundidas ante textos complejos que en la mayoría de los casos necesitan guías detalladas que les ayuden en sus primeros pasos cuando se trata de adoptar DDD. Por esto, aquellos que usan DDD repasan la literatura de referencia varias veces. De hecho, incluso se podría llegar a la conclusión que nunca se sabe todo; por ello, este libro de alguna manera servirá como referencia rápida y, a medida que mejore la experiencia, será necesario usar otras obras para profundizar en otros detalles. Muchas personas han tenido problemas para vender DDD a sus colegas o al siempre importante equipo directivo. Este libro ayudará a hacerlo, no solo al explicar DDD en forma resumida, sino asimismo porque enseña las herramientas disponibles para acelerar su adopción y gestionar su uso.

Evidentemente, en este libro no es posible enseñar todo sobre DDD, ya que se han sintetizado a propósito las técnicas de DDD. Para profundizar más, se puede consultar *Implementing Domain-Driven Design*, y se recomienda el taller *Cómo implementar DDD* (<https://idddworkshop.com>) en tres días. Ese curso intensivo de tres días, impartido por todo el mundo a multitud de desarrolladores de naturaleza muy variada, ayudará a una actualización inmediata de DDD. También hay formación sobre DDD para usuarios en linea en <https://Kalele.io>.

Las buenas noticias son que DDD no tiene por qué convertirse en una pesadilla. Así como muy probablemente haya que lidiar con la complejidad en los proyectos, también es posible aprender a usar DDD para superar circunstancias difíciles.

## Diseños buenos, malos y efectivos

---

A menudo se habla de buenos o malos diseños. Y surge el interrogante, ¿qué tipo de diseño se hace? Muchos equipos de desarrollo de software no dedican casi nada de tiempo a esta tarea. En su lugar, practican lo que se conoce como “organizar las tareas”. Es decir, un equipo tiene una lista de tareas en desarrollo, con el Backlog del proyecto en el contexto de Scrum, y pasan la nota adhesiva de su tablero de la columna “Pen-

dientes” a la columna “En proceso”. Y ahí acaban todas las intenciones y reflexiones previas para hacer el trabajo correspondiente. No hay diseño. El resto se deja a criterio de los programadores quienes se dedican a escribir sin pensar el código fuente correspondiente. Por ello, rara vez sale tan bien como debería, y el costo para el negocio que paga por estos diseños inexistentes por lo general es muy alto.

Esto sucede a menudo debido a la presión de producir nuevas versiones de software siguiendo un riguroso plan de fechas y calendario, donde la gerencia utiliza Scrum para controlar las entregas en lugar de posibilitar uno de los principios más importantes de Scrum: *adquisición de conocimiento*.

Cuando las empresas acuden a una consultoría o a procesos de capacitación, generalmente se presentan las mismas situaciones. Proyectos de software en peligro y equipos muy grandes que reparan con retazos de código y datos para tratar de mantener los sistemas en funcionamiento. A continuación, se presentan los problemas más complejos en estas instancias y que curiosamente DDD puede ayudar a evitar fácilmente a los equipos; en seguida, las dificultades relacionadas con negocios de más alto nivel, luego las de carácter técnico:

- El desarrollo de software se considera un centro de costos, en vez de asumirlo como un generador de utilidades. Por lo general, esto se debe a que la organización percibe los computadores y el software como males necesarios para la empresa, en lugar de tomarlos como fuentes de su ventaja estratégica. (Infortunadamente, puede que no haya cura para esto ya que es algo muy arraigado en la cultura empresarial actual).
- Los desarrolladores están demasiado obsesionados con la tecnología e intentan resolver los problemas al usar más tecnología, en lugar de dedicarse a pensar y diseñar primero. Esto los lleva a perseguir constantemente “objetos deslumbrantes” que aparecen con la última moda tecnológica.
- La base de datos es prioridad absoluta. La gran mayoría de discusiones para resolver problemas técnicos se concentran en la base de datos y en su diseño, en lugar de trabajar en los procesos y operaciones del negocio.
- Los desarrolladores no hacen suficiente énfasis en usar una nomenclatura para objetos y componentes acordes con la función del negocio

que asumen. Esto produce un gran abismo entre el modelo mental de la empresa y el software que proporcionan los desarrolladores.

- El problema anterior generalmente es resultado de una colaboración pobre con el negocio. A menudo, los interesados de la organización gastan mucho tiempo y producen aisladamente especificaciones utilizadas apenas parcialmente por parte de los desarrolladores o a la postre simplemente no las aprovechan.
- El negocio demanda muchas estimaciones y con el máximo de precisión posible. Calcularlas consume mucho tiempo y esfuerzo y, a menudo, se generan retrasos en los entregables de software. Los desarrolladores practican lo que se conoce como “organizar las tareas”, en vez de concentrarse en un diseño concienzudo. Crean una “*Big Ball of Mud*” (gran bola de lodo), tema por analizar en los siguientes capítulos, en vez de segmentar adecuadamente los modelos según las áreas del negocio.
- Los desarrolladores ven la lógica del negocio en los componentes de la interfaz de usuario y en mecanismos de persistencia. Además, con frecuencia ejecutan operaciones de persistencia en medio de la lógica del negocio.
- Las consultas en las bases de datos aparecen fragmentadas, son lentas o producen bloqueos, lo que impide a los usuarios realizar operaciones de negocios donde el factor tiempo es importante.
- Asimismo, se han creado abstracciones innecesarias en las que los desarrolladores intentan resolver todas las necesidades actuales y futuras, imaginables e inimaginables, al generalizar exageradamente las soluciones, en lugar de abordar las necesidades concretas del negocio.
- Existen servicios muy acoplados entre sí, donde una operación que se ejecuta dentro de un servicio acaba comprometiendo directamente a otro servicio para delegar parte de las responsabilidades. Este acoplamiento conduce a procesos empresariales que normalmente no funcionan bien o contienen datos inconsistentes, sin mencionar aquellos sistemas muy difíciles de mantener.

Todo eso parece ocurrir en la filosofía de “no diseñar produce software a menor costo”. Y con mucha frecuencia simplemente es una cuestión

entre la organización y los desarrolladores de software que no saben que existe una alternativa mucho mejor. “El software se está comiendo al mundo” [Wall Street Journal], y lo que en realidad debería importar es que el software también puede acabar con las utilidades o incrementarlas en gran medida.

De igual modo, resulta importante tener claro que la mágica economía de *no diseñar* constituye una falacia que ha engañado a todos aquellos que presionan a los equipos a producir software sin llevar a cabo un proceso de diseño cuidadoso. Esto se debe a que todavía se considera que el diseño está en el cerebro de los desarrolladores que espontáneamente manipulan códigos fuente con la magia de sus dedos, sin tener en cuenta información esencial de los demás, incluida la misma organización. Esta cita de Douglas Martin resume muy bien esta idea:

Los cuestionamientos sobre si el diseño es necesario o asumible en cuestión de costos están fuera de lugar: en realidad, el diseño es un tema inevitable. La alternativa a un buen diseño es lo opuesto, sencillamente algo que no sirve, no la decisión de no diseñar. *Book Design: A Practical Introduction*

Aunque los planteamientos de Martin no son específicamente sobre diseño de software, son fácilmente aplicables al oficio, pues no hay sustituto para un diseño cuidadoso. En la situación que se acaba de exponer, si hay cinco desarrolladores de software trabajando en un proyecto, la alternativa de *no diseñar* producirá una amalgama de cinco diseños diferentes en uno solo. Es decir, se obtiene una combinación de cinco interpretaciones inventadas y diferentes del lenguaje del negocio que se desarrollan sin contar con el beneficio de los *expertos de dominio* reales.

El resultado final: se modela sí o sí, independientemente de que se sea consciente o no de ello. Esto puede compararse a la manera como se han construido las carreteras. Algunos caminos antiguos empezaron como senderos para carretas que finalmente se convirtieron en rutas muy transitadas; tomaron giros inexplicables y se bifurcaron de tal modo que solo servirían a unos pocos con necesidades muy rudimentarias. En algún momento, estos caminos fueron pavimentados para la comodidad de un número de viajeros cada vez mayor que los usaban. Estas vías de paso improvisadas no se siguen usando en la actualidad, porque estuviesen bien diseñadas sino porque ya existían. Pocas personas contemporáneas pueden comprender por qué viajar por una de estas vías es tan incómodo.

do e inconveniente. Las carreteras modernas se planifican y diseñan de acuerdo con estudios cuidadosos sobre la población, el medio ambiente y un flujo predecible. Ambos tipos de caminos son modelados; en un modelo se empleó una reflexión mínima mientras en el otro se explota el máximo conocimiento. Así mismo, el software se puede modelar desde cualquier perspectiva.

Si se cree que desarrollar software con un diseño inteligente es costoso, hay que pensar más bien en cuán costoso será vivir con él o incluso corregir un diseño deficiente. Esto es especialmente relevante cuando se habla de software que necesita constituir un factor diferenciador entre una organización y las demás, así como de generar considerables ventajas competitivas.

Una palabra estrechamente relacionada con bueno es *efectivo*, y esto posiblemente indique con mayor precisión qué se debe buscar en el diseño de software: un diseño *efectivo*. Un diseño de esta naturaleza satisface las necesidades de la organización empresarial en la medida en que puede distinguirse de la competencia por el software; de hecho, obliga a la organización a comprender en qué debe sobresalir y se utiliza para orientar la creación de un modelo correcto de software.

En Scrum, la *adquisición de conocimiento* se hace a través de la experimentación y el aprendizaje colaborativo. A ese concepto se le conoce como “comprar información” *Essential Scrum* (Scrum esencial). El conocimiento nunca es gratuito, y este libro en esa dirección ofrece múltiples maneras de obtenerlo.

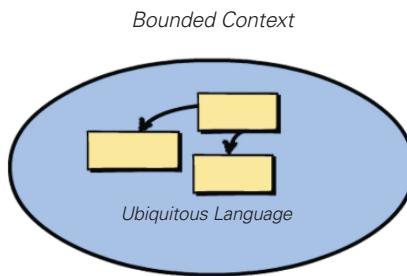
Solo por si acaso todavía hay dudas que importa tener un diseño efectivo, no se deben olvidar los descubrimientos de alguien que parece haber comprendido su real importancia:

La mayoría de las personas cometen el error de creer que el diseño es lo que se ve. La gente cree en esa apariencia, y piensa que a los diseñadores se les entrega una caja y se les dice: “¡Hagan que se vea bien!” Eso no es en verdad lo que creemos que es el diseño. No es solo lo que parece y se siente. El diseño consiste en ver cómo funciona.

—Steve Jobs

En software, un diseño efectivo es lo que más importa y es la única alternativa recomendable.

## Diseño estratégico



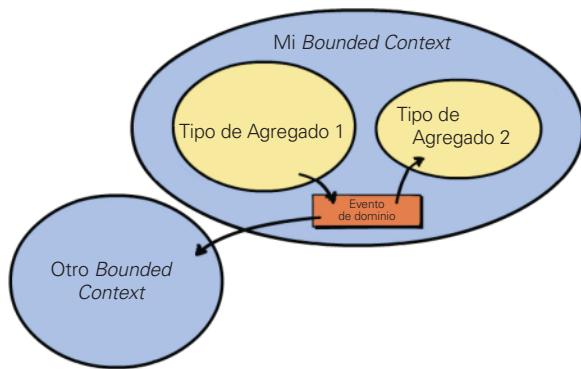
Para comenzar, el diseño estratégico es de vital importancia. La verdad es que no es posible aplicar efectivamente el diseño táctico a menos que se empiece con el diseño estratégico. Este se usa como grandes pinceladas antes de entrar en los detalles de la implementación. Resalta lo que es estratégicamente importante para un negocio, cómo desglosar el trabajo por importancia y cómo integrar lo mejor en la medida de lo necesario.

Primero, es necesario aprender a segregar los modelos de dominio utilizando el patrón de diseño estratégico denominado *Bounded Context* (contexto delimitado). Al mismo tiempo, se podrá ver cómo desarrollar un *Ubiquitous Language* como modelo del dominio dentro de un *Bounded Context* explícitamente definido.

Asimismo, es importante aprender a interactuar no solo con desarrolladores, sino también con *expertos de dominio* a medida que se desarrolla el *Ubiquitous Language* de un modelo. Así, se verá cómo trabajan colaborativamente los desarrolladores de software y los expertos de dominio. Esta es una combinación vital de personas inteligentes y motivadas que son necesarias para que DDD produzca los mejores resultados posibles. El lenguaje que desarrollen juntos en equipo se convertirá en ubicuo para todo el modelo de software y para la comunicación del equipo.

A medida que se avance en el diseño estratégico, se aprenderá sobre los subdominios y cómo estos pueden ayudar a lidiar con la complejidad ilimitada de los sistemas legados, además de cómo mejorar los resultados en *Greenfield Projects* (proyectos que inician de cero). También se verá cómo integrar varios *Bounded Contexts* mediante una técnica llamada *Context Mapping* (mapeo entre contextos). Los *Context Maps* (mapas de contexto) definen tanto las relaciones entre equipos como los mecanismos técnicos que existen entre dos *Bounded Contexts*.

## Diseño táctico



Después de contar con una base sólida para el diseño estratégico, el usuario descubrirá las herramientas de diseño táctico más importantes de DDD. El diseño táctico es como usar un pincel fino para trazar los detalles del modelo de dominio. Una de las herramientas más importantes se utiliza para agregar *entities* (entidades) y *value objects* (objetos de valor) juntos en un único clúster del tamaño apropiado. Es el *Aggregate pattern* (patrón agregado).

DDD tiene que ver con modelar un dominio de la forma más explícita posible. El uso de *Domain Event* (evento de dominio) ayudará tanto a modelar explícitamente como a compartir lo ocurrido dentro del modelo con los sistemas que necesiten un reconocimiento de este. Los interesados pueden ser otros *Bounded Contexts* remotos o incluso el propio *Bounded Context* local.

## El proceso de aprender y destilar conocimiento



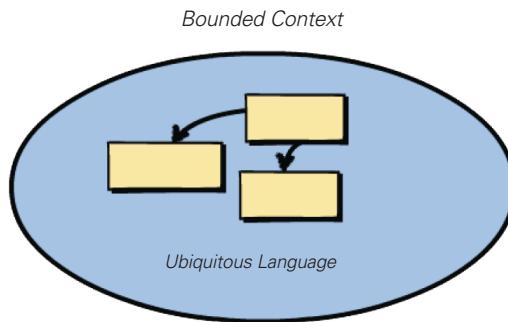
DDD enseña una manera de pensar para ayudar al usuario y al equipo a mejorar sus conocimientos a medida que aprende las competencias básicas de la empresa. Este proceso de aprendizaje es cuestión de descubrimiento a partir del diálogo y la experimentación en grupo. Cuando se cuestiona el *statu quo* y se desafían supuestos de un modelo de software, el aprendizaje será enorme y esta adquisición importante de conocimiento se extenderá por todo el equipo. Esta es una inversión decisiva para el negocio y el equipo. El objetivo debe ser no solo aprender y refinar, sino también aprender y destilar lo más rápido posible. Existen otras herramientas para ayudar con esos objetivos que se pueden encontrar en el capítulo 7, “Herramientas de gestión y aceleración”.

### ¡Hora de comenzar!

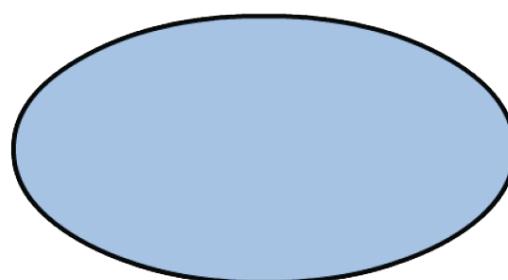
Incluso en una presentación resumida, hay mucho que aprender sobre DDD. Así que hay que continuar con el capítulo 2, “Diseño estratégico con *Bounded Contexts* y *Ubiquitous Language*”.

## CAPÍTULO 2

# Diseño estratégico con *Bounded Contexts* y *Ubiquitous Language*



¿A qué responde esto que se denomina *Bounded Contexts*? ¿Qué es el *Ubiquitous Language*? En resumen, DDD tiene que ver principalmente con modelar de forma explícita un *Ubiquitous Language* dentro de un *Bounded Context*. Es cierto que probablemente por el momento no sea la descripción más útil del mundo para hacer llegar al usuario, por ello es mejor desarrollar un poco más el concepto.



Para empezar, un *Bounded Context* es un límite contextual semántico. Esto significa que, dentro de un espacio delimitado, cada componente del modelo de software tiene un significado específico que desarrolla cosas específicas. Los componentes dentro de un *Bounded Context* son específicos del contexto y su motivación es totalmente semántica. Así de sencillo.

Cuando alguien empieza a modelar software, su *Bounded Context* es en cierta medida conceptual. Podría considerarlo como parte del problema por resolver, lo que se conoce como el espacio del problema. Sin embargo, a medida que ese modelo comience a tener más claridad y definición, el *Bounded Context* pasará rápidamente a ser parte de la solución, lo que se conoce más específicamente como espacio de la solución, y el modelo de software se irá reflejando en el proyecto como código fuente. El espacio del problema y el espacio de la solución se explican mejor en el siguiente recuadro. Recuérdese que un *Bounded Context* es el lugar donde se implementa un modelo y habrá una serie de artefactos de software diferentes por cada *Bounded Context*.

---

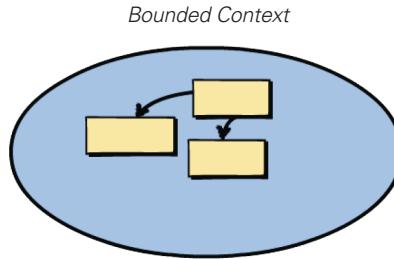
### ¿Cómo definir los espacios del problema y de la solución?

El espacio del problema es donde se realiza el análisis estratégico de alto nivel y los primeros pasos en el diseño, dentro de las restricciones del proyecto. Se pueden usar diagramas sencillos a medida que se analizan los direccionadores estratégicos del proyecto y se anotan los riesgos y objetivos más importantes. En la práctica, los *Context Maps* funcionan muy bien en el espacio del problema. Obsérvese también que los *Bounded Contexts* se pueden usar en los diferentes análisis sobre el espacio del problema, si es necesario, aunque están estrechamente relacionados con el espacio de la solución.

El espacio de la solución es donde realmente se implementa una salida. Es el resultado de los análisis que hayan ocurrido sobre el espacio del problema y que han identificado el *Core Domain* (dominio núcleo). Cuando se desarrolla un *Bounded Context* como iniciativa estratégica clave de una organización, se conoce como *Core Domain*. El *Bounded Context* se desarrolla como código, tanto fuente principal como fuente de prueba. Dentro del espacio de la solución, también se escribe un código que soporta la integración con otros *Bounded Contexts*.

---

## Cómo definir los espacios del problema y de la solución



El modelo de software inmerso en un límite contextual refleja un lenguaje desarrollado por el equipo que trabaja en el *Bounded Context*. Ese lenguaje se habla por parte de cada miembro del equipo que crea dicho modelo de software y que funciona dentro de ese *Bounded Context*.

El lenguaje se conoce como *Ubiquitous Language* porque se habla entre todos los miembros del equipo y se implementa en el modelo de software. Por tanto, es necesario que el *Ubiquitous Language* sea riguroso: estricto, exacto y preciso. En la figura se muestran los cuadros dentro del *Bounded Context*, que representan los conceptos del modelo que pueden implementarse como clases. Cuando el *Bounded Context* se desarrolla como iniciativa estratégica clave de una organización, se llama *Core Domain*.

Cuando se compara con todo el software que usa una organización, un *Core Domain* constituye un modelo de software que se encuentra entre los más importantes, porque es un medio para alcanzar la excelencia. Un *Core Domain* se desarrolla para distinguir competitivamente una organización de todas las demás y, como mínimo, se asocia a alguna de las líneas de negocios más importantes. Una organización no puede sobresalir en todo y ni siquiera debería intentarlo. Así que es aconsejable escoger lo que debería y no debería ser parte de un *Core Domain*. Ésta es la propuesta de valor principal de DDD, para que se haga una inversión inteligente y asignar los mejores recursos a un *Core Domain*.



Cuando alguien en un equipo usa expresiones propias del *Ubiquitous Language*, todos los miembros del grupo entienden con precisión lo que quiere decir. Entonces, la expresión se hace ubicua en el seno del equipo, al igual que el lenguaje utilizado que define el modelo de software que se desarrolla.

Cuando alguien considera el lenguaje en un modelo de software, piensa en los distintos países que conforman Europa. En cada uno de los países del viejo continente, el idioma oficial de cada país está debidamente definido. Dentro de los límites de cada uno de los países, por ejemplo, Alemania, Francia e Italia, los idiomas oficiales están claramente definidos, pero al cruzar cada frontera, la lengua oficial cambia. Lo mismo ocurre con Asia, donde se habla japonés en Japón y donde los idiomas que se hablan en China y Corea son evidentemente diferentes al cruzar sus respectivas fronteras. Por ello, se puede pensar en los *Bounded Contexts* de la misma forma como fronteras idiomáticas. En el caso de DDD, los lenguajes son aquellos idiomas que habla cada equipo, que posee un modelo de software. Hay que destacar entonces que existe una importante representación escrita de los lenguajes de esos modelos de software, que es el código fuente. El *Bounded Context*, los equipos y repositorios de código fuente se detallan en el siguiente recuadro.

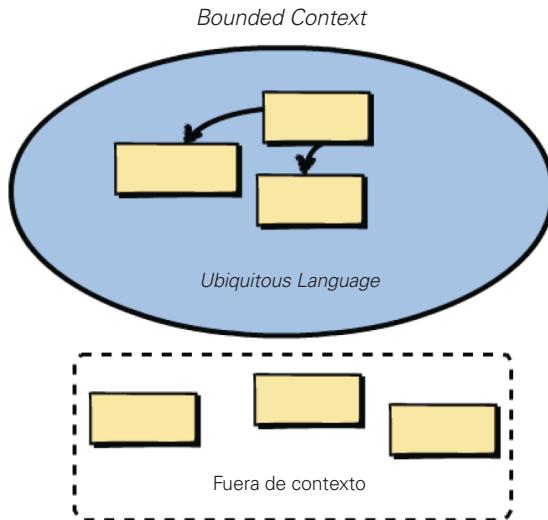
---

### ***Bounded Context, equipos y repositorios de código fuente***

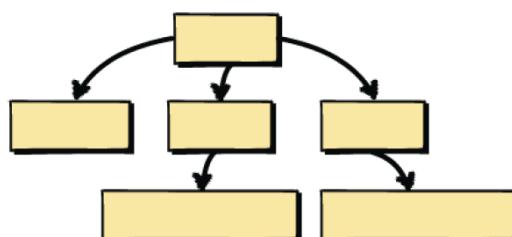
En un *Bounded Context*, un solo equipo debería estar asignado para trabajar. También debería existir un único repositorio de código fuente por cada *Bounded Context*. Es posible que un equipo pueda trabajar en múltiples *Bounded Contexts*, pero varios equipos no deberían trabajar en un solo *Bounded Context*. Separar claramente el código fuente y el esquema de la base de datos para cada *Bounded Context* de la misma manera que separa el *lenguaje ubicuo*. Es aconsejable entonces mantener tests de aceptación y los tests unitarios junto con el código fuente principal.

Asimismo, es especialmente importante tener claro que un equipo trabaja en un único *Bounded Context*. Esto elimina completamente la posibilidad de tener sorpresas ingratas que surjan cuando otro equipo realice un cambio en el código fuente. El equipo posee el código fuente y la base de datos; igualmente define las interfaces oficiales mediante las cuales se debe usar un *Bounded Context*. Es uno de los beneficios de usar DDD.

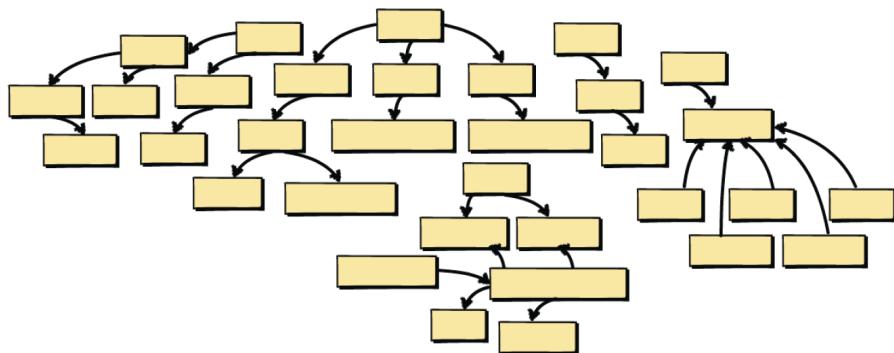
---



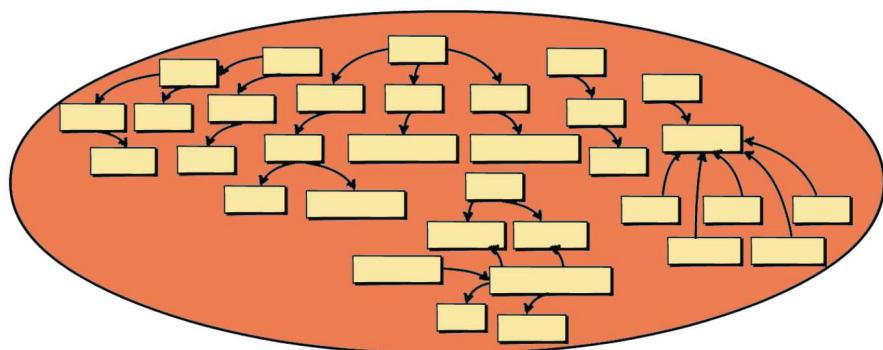
En los idiomas, el vocabulario evoluciona con el tiempo y atraviesa las fronteras entre países y culturas. Estos factores, y muchos otros, hacen que las mismas palabras o similares adquieran diferentes matices en sus significados. Obsérvense las diferencias que hay entre las palabras en español utilizadas en España y las mismas en Colombia, donde incluso la pronunciación y la acentuación cambian. Evidentemente, hay un español de España y un español con matices colombianos. Esto también ocurre con los lenguajes de los modelos de software. Es posible que para un mismo término, las personas de diferentes equipos tengan una definición diferente, y esto se debe a que su conocimiento del negocio está relacionado con un contexto distinto, ya que ellos trabajan en un *Bounded Context* diferente. No se espera entonces que ningún componente por fuera de determinado contexto se adhiera a las mismas definiciones; de hecho, es probable que sean ligera o completamente diferentes, y eso está bien.



Para comprender uno de los grandes motivos para usar *Bounded Contexts*, obsérvese un problema común en el diseño de software. A menudo, los equipos no saben cuándo dejar de acumular más y más conceptos en sus modelos de dominio. El modelo puede comenzar pequeño y manejable...



Pero luego el equipo agrega cada vez más conceptos casi sin parar, tanto que se convierte en un gran problema. No sólo hay demasiados conceptos, sino que el lenguaje del modelo se hace impreciso, y cuando menos se piensa, en realidad hay varios lenguajes coexistiendo en un mismo modelo sobredimensionado, confuso y sin límites.



Debido a este error, los equipos a menudo convierten un producto de software completamente nuevo en lo que se conoce como *Big Ball of Mud*. Sin duda, llegar a esa instancia no es algo para sentirse orgulloso;

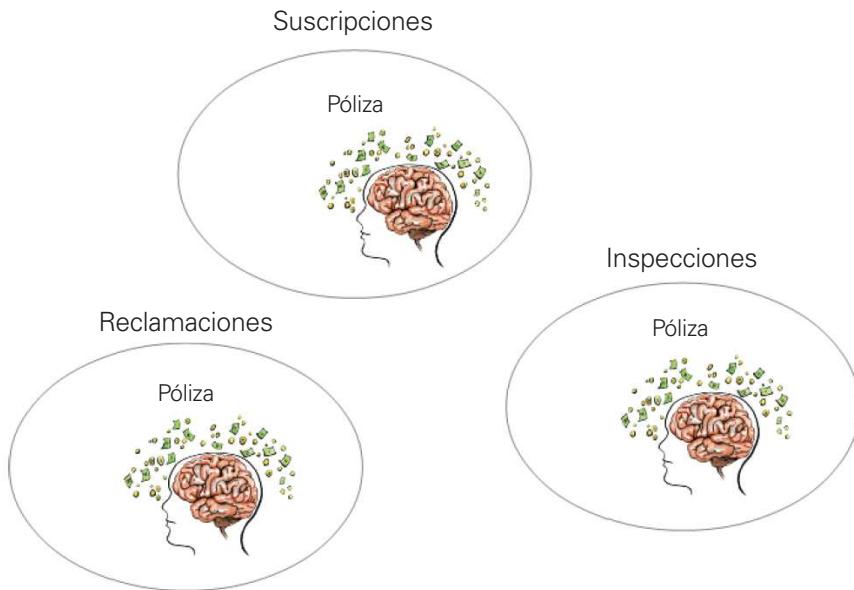
es un monolito, o incluso algo peor. Aquí es donde un sistema se enreda con múltiples modelos sin delimitarlos explícitamente. Probablemente también requiera que varios equipos trabajen en el mismo sistema, lo cual se hace muy problemático. Además, varios conceptos que no estén relacionados se pueden filtrar a través de muchos módulos e interconectarse con elementos generando conflictos entre sí. Si este proyecto tiene tests, probablemente ejecutarlos demande mucho tiempo y, en momentos críticos, se acabe por obviarlos.

Es el resultado de tratar de hacer demasiadas cosas, con demasiadas personas y en el lugar equivocado. Cualquier intento de desarrollar y hablar un *Ubiquitous Language* producirá un dialecto fragmentado y mal definido que pronto será abandonado. El lenguaje ni siquiera se consideraría una forma de esperanto; sólo sería un desastre, como una *Big Ball of Mud*.

## Expertos del dominio e impulsores de negocios



Puede que los responsables del negocio hayan sugerido ideas clave, o incluso otras más sutiles, que el equipo técnico podría haber utilizado para tomar mejores decisiones de modelación. Por tanto, a menudo el resultado de un esfuerzo desenfrenado es una *Big Ball of Mud*, por parte de un equipo de desarrolladores de software que no escuchan a los expertos del negocio.



Los diferentes departamentos de la organización o grupos de trabajo existentes, entonces, pueden ser un buen indicador de dónde deberían existir los límites del modelo. Se tiende así a buscar al menos un experto del negocio por función del negocio. Hoy en día, hay tendencia a agrupar personas por proyecto, mientras las divisiones del negocio o incluso los grupos funcionales bajo jerarquía organizativa parecen ser menos populares. Incluso frente a los modelos de negocio más modernos, se encuentra que los proyectos están organizados de acuerdo con los direcccionadores del negocio y bajo áreas de experticia; por ello, es posible pensar en la división o función en esos términos.

Por consiguiente, se puede determinar que este tipo de segregación es necesaria si se tiene en cuenta que es probable que cada función del negocio tenga definiciones diferentes para el mismo término. Considérese el concepto denominado “póliza” y cómo el significado difiere entre las diversas funciones de una aseguradora. Es posible imaginar fácilmente que una póliza para el departamento de suscripciones es muy diferente de una póliza para la división de reclamaciones o incluso para la sección de inspecciones. Obsérvese el recuadro para más detalles.

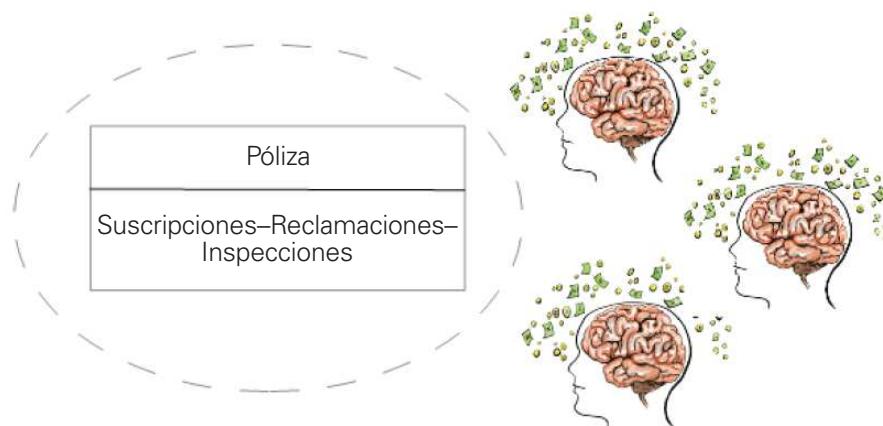
En cada una de estas áreas de negocio, la existencia de la póliza varía por diferentes razones. No hay escapatoria a este hecho, es intrínseco del sector y no existe perspectiva que lo cambie.

## Diferencias en las pólizas por función

Póliza en suscripción: en el departamento especializado en suscripciones, se crea una póliza basada en la evaluación de riesgos de la entidad asegurada. Por ejemplo, al trabajar en la suscripción de seguros de propiedades, los expedidores evaluarían los riesgos asociados a determinada propiedad para calcular la prima de la póliza que cubra el activo de la propiedad.

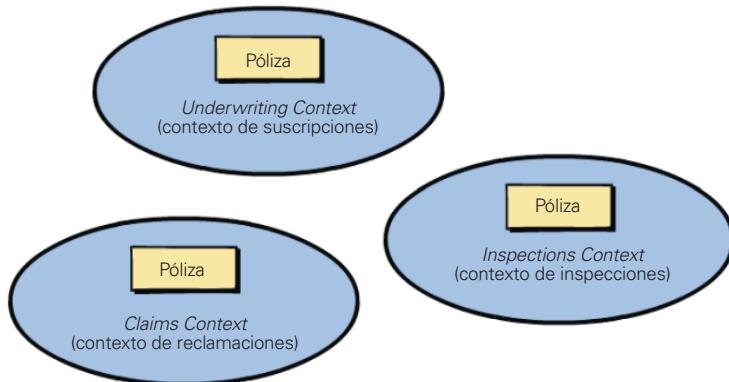
Póliza en inspecciones: nuevamente, si se trabaja en el campo del seguro de propiedades, la aseguradora probablemente tendrá un equipo de peritos responsables de inspeccionar la propiedad por asegurar. Los expedidores dependen en alguna medida de la información hallada durante las inspecciones desde la perspectiva que la propiedad está en la condición declarada por el asegurado. Suponiendo que una propiedad se vaya a asegurar, los detalles de la inspección (fotos y observaciones) se vinculan a una póliza en el área de inspecciones, y sus datos pueden ser referenciados por el departamento de suscripciones para negociar el costo de la prima final en dicho departamento.

Póliza en reclamaciones: una póliza en este departamento comprueba las solicitudes de cobro de los asegurados en función de los términos de la póliza creada por el área de expediciones. La póliza de reclamaciones debe hacer referencia a la póliza suscrita, pero se concentrará, por ejemplo, en daños a la propiedad asegurada y las revisiones realizadas por el personal de reclamaciones para determinar el pago, si corresponde, que debe realizarse.



Entonces, si se intenta unificar los tres tipos de pólizas en una sola para dar respuesta a las necesidades de las tres áreas del negocio, segu-

ramente habrá problemas. Y seguramente la situación empeorará si la póliza, que ya contiene mucha lógica, tuviera que dar soporte en el futuro a un cuarto o quinto concepto de negocio. Y nadie saldría ganando.



Por otro lado, DDD hace énfasis en todas esas diferencias desglosando los diversos tipos de pólizas en distintos *Bounded Contexts*. Hay que reconocer que existen diferentes lenguajes y funciones, lo que lleva a actuar en consecuencia. ¿Hay tres significados para la póliza? Luego, existen tres *Bounded Contexts*, cada uno con su propia póliza, y cada una con sus propias características que las hacen únicas. No es necesario llamar a estas pólizas (*Policy*), bien sea *UnderwritingPolicy*, *ClaimsPolicy*, o *InspectionsPolicy*. El nombre del *Bounded Context* ya se encargará de resolver ese problema. El nombre es simplemente *póliza* para cada uno de los tres *Bounded Contexts*.

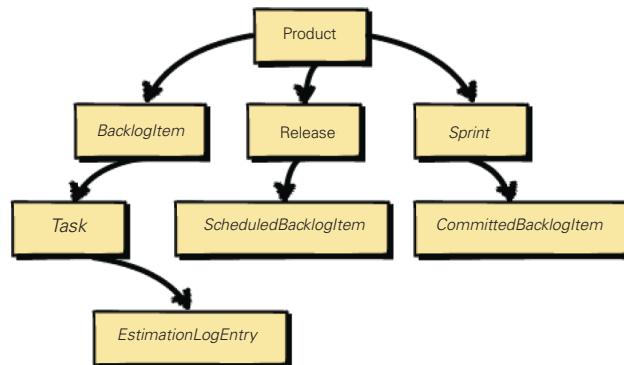
---

### Otro ejemplo: ¿qué es un vuelo?

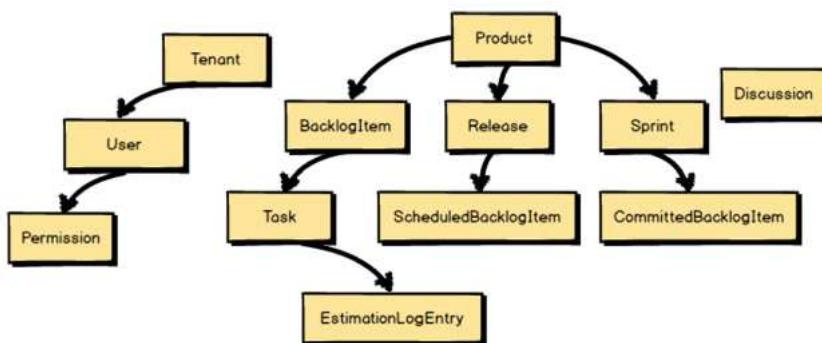
En la industria del transporte aéreo, un “vuelo” puede tener múltiples significados. Un vuelo se puede definir como un único despegue y aterrizaje en el que la aeronave vuela de un aeropuerto a otro. También se puede usar para hacer referencia al mantenimiento de las aeronaves. Otra definición hace referencia a la emisión de tiquetes de pasajeros, bien sea con una o ninguna escala. Debido a que cada uno de estos usos de “vuelo” se entiende con claridad sólo por contexto, cada uno debe ser modelado en un *Bounded Context* separado. Modelar los tres en el mismo *Bounded Context* generaría confusión.

---

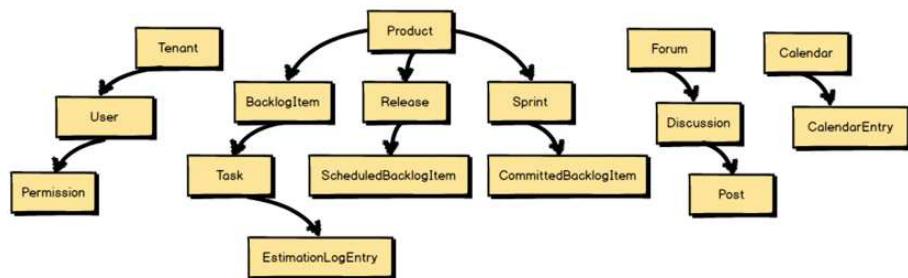
## Caso de estudio



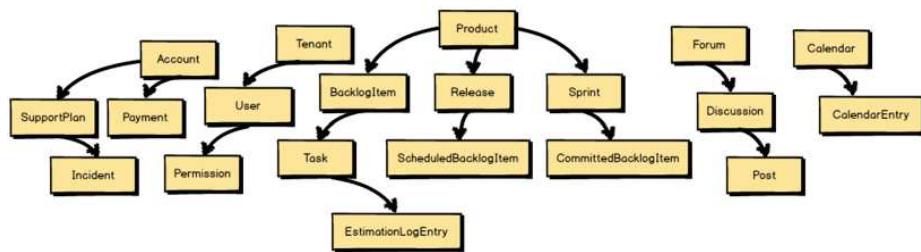
Para concretar mejor los beneficios de usar múltiples *Bounded Contexts*, se puede ilustrar con una muestra de modelo de dominio. En este caso, se puede trabajar con una aplicación de gestión de proyectos ágiles basada en Scrum. Así, un concepto fundamental es el **producto**, que representa el software por construir y perfeccionar a lo largo de años de desarrollo. Este producto tiene a su vez un conjunto de *Items del Backlog*, Releases y *Sprints*. Cada *Item del Backlog* por completar contiene un conjunto de tareas. Cada tarea puede contener a su vez una colección de Estimation Log Entries. Las *Releases* tienen *Scheduled Backlog Items* para cierta fecha, y los *Sprints* tienen *Committed Backlog Items*. Hasta aquí, todo está claro, ya que se identifican los conceptos principales del modelo de dominio y el lenguaje es claro y conciso.



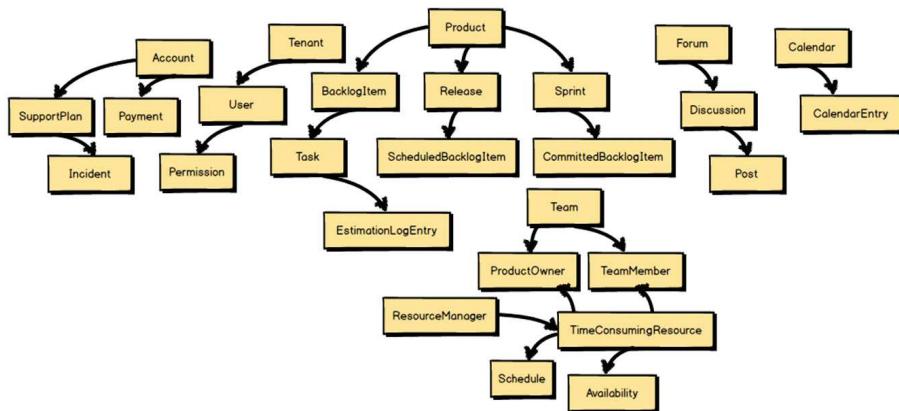
“Ah, es cierto”, dicen los miembros del equipo, “también necesitamos nuestros propios usuarios. Y no olviden que queremos facilitar análisis colaborativos dentro del equipo de producto. Representemos a cada organización que se suscriba como un *Tenant* (tenedor) que tiene su propio espacio. Dentro de un *Tenant*, permitiremos el registro de un número de *Users* (usuarios); estos también tendrán *Permissions* (permisos). Y agregaremos un concepto llamado *Discussion* (discusión) para representar una de las herramientas de colaboración a la cual daremos soporte”.



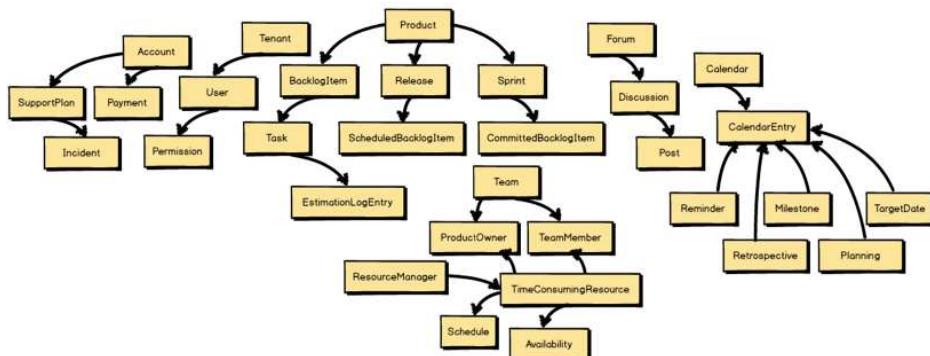
Paso seguido, los miembros del equipo agregan: “Bueno, también hay otras herramientas de colaboración. *Discussions* pertenecen a los foros y tienen *Posts* (entradas). También queremos soportar *Shared Calendars* (calendarios compartidos)”.



Y siguen anotando: “Y no olviden que necesitamos una forma para que cada *Tenant* pueda realizar *Payments* (pagos). También venderemos diferentes planes de soporte por niveles, por ello necesitamos una forma de rastrear los reportes; tanto el tipo de soporte como los pagos deben ser administrados bajo una misma *Account* (cuenta)”.

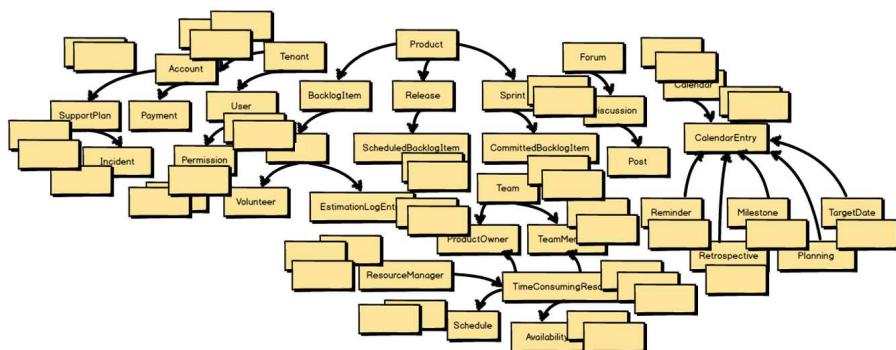


Y todavía más conceptos: “Cada producto basado en Scrum tiene un *equipo* específico que trabaja en dicho producto. Los equipos están compuestos por un único *Product Owner* (propietario del producto) y varios *Team Members* (miembros de equipo). Entonces, ¿cómo podemos abordar todo lo relacionado con la *utilización de recursos humanos*? ¿Y qué pasaría si modelamos los horarios de los miembros de equipo junto con su *uso y disponibilidad*? ”



“¿Saben qué más?”, cuestionan, “los *Shared Calendars* (calendarios compartidos) no deben limitarse a *Calender Entries* (entradas de calendario genéricas). Deberíamos identificar tipos específicos de *Shared Calendars*, como *Reminders*, *Milestones* (hitos de equipos), reuniones de planificación y retrospectivas, y fechas objetivo”. ¡Pero, un momento! ¿Ven la trampa en la que está cayendo el equipo?

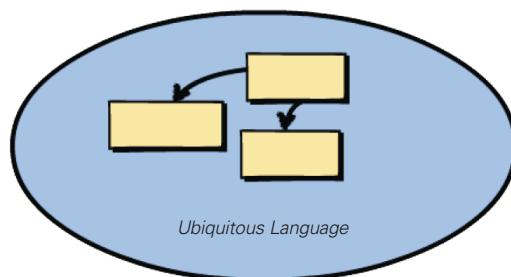
Observemos qué tanto se han alejado de los conceptos centrales originales de producto, *Backlog Items*, *Releases* y *Sprints*. El lenguaje ya no es exclusivamente sobre Scrum, se ha fracturado y confundido.



El número de conceptos que han aparecido no son muchos pero no hay que dejarse engañar. Para cada uno de ellos, podrían venir a la mente dos o tres conceptos adicionales que dependan de los que ya se tienen sobre la mesa. El equipo ya está camino de entregar una *Big Ball of Mud* y el proyecto apenas comienza.

## Diseño estratégico fundamental requerido

*Bounded Context*



¿Cuáles herramientas están disponibles con DDD para ayudar a evitar tales riesgos? Se necesitan al menos dos herramientas fundamentales de diseño estratégico. Una es el *Bounded Context* y otra, el *Ubiquitous Language*. El uso de un *Bounded Context* obliga a responder

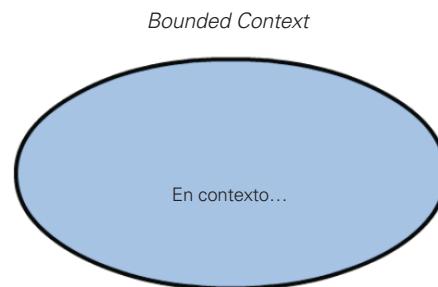
la pregunta “¿Qué es fundamental?” El *Bounded Context* debe mantener juntos todos los conceptos que son fundamentales para la iniciativa estratégica y dejar fuera todo lo demás. Los conceptos que quedan forman parte del *Ubiquitous Language* del equipo. Así se verá cómo funciona DDD para evitar el diseño de aplicaciones monolíticas.

---

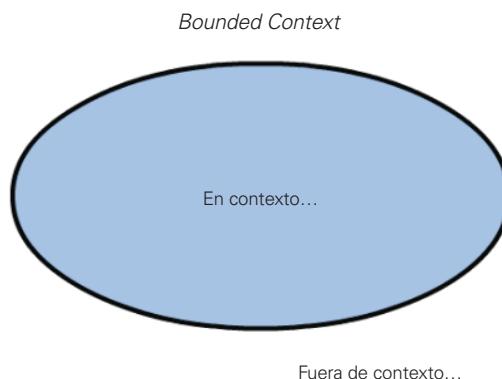
### Beneficios al hacer pruebas

Debido a que los Bounded Contexts no son monolíticos, se pueden experimentar otros beneficios colaterales al utilizarlos. Uno de estos es que los tests se centrarán en un único modelo y, por tanto, serán menos numerosos y se ejecutarán más rápidamente. Si bien esta no es la motivación principal para usar Bounded Contexts, con seguridad contribuye positivamente.

---



Literalmente, algunos conceptos estarán dentro del contexto y se incluirán evidentemente en el lenguaje del equipo.



Y otros conceptos estarán fuera del contexto. Los conceptos que sobreviven por aplicar rigurosamente el filtro de lo que es fundamental o no, forman parte del *Ubiquitous Language* del equipo que es dueño del *Bounded Context*.

### Tome nota

Los conceptos que sobreviven por aplicar rigurosamente el filtro de lo que es fundamental o no, forman parte del *Ubiquitous Language* del equipo que es dueño del *Bounded Context*. Establecer un límite obliga a ser riguroso con los conceptos que quedan en el interior.



Entonces, ¿cómo saber qué es fundamental? Aquí es donde es imprescindible reunir dos grupos vitales de individuos en un equipo altamente cohesionado y colaborativo: *expertos de dominio* y *desarrolladores* de software.



Los *expertos de dominio* estarán naturalmente más enfocados en los intereses del negocio. Sus pensamientos se concentrarán en la visión de cómo funciona el negocio. En el caso del dominio de Scrum, cuenta con que el *experto de dominio* sea un Scrum Master que comprende perfectamente cómo se implementa y funciona Scrum en un proyecto.

### *¿Product Owner o experto de dominio?*

Es posible que surja el interrogante sobre cuál es la diferencia entre un *Product Owner Scrum* y un experto de dominio en DDD. Bueno, en algunos casos pueden ser la misma y única persona, es decir, alguien en capacidad de cumplir ambos roles a la vez. Sin embargo, no debe sorprender que el *Product Owner* por lo general esté más enfocado en administrar y priorizar el listado de tareas pendientes y ver que se mantenga la continuidad conceptual y técnica del proyecto. Sin embargo, esto no significa que el *Product Owner* sea naturalmente un experto en la competencia clave de la empresa en la que alguien está trabajando. Hay que asegurarse de que existe un verdadero *experto de dominio* en el equipo y no usar por defecto al *Product Owner* que puede no tener el conocimiento necesario.

En una empresa, seguramente también existen *expertos de dominio*, que no es un cargo oficial sino describe a aquellos que se concentran principalmente en el negocio. Es con su modelo mental con el que empieza a formarse la base del *Ubiquitous Language* del equipo.

```
while (a < b) { 0xFB249E7
    a += c;          C# Scala Java
}
                JavaScript PHP   BPM
10110001101010110001 BPEL
```



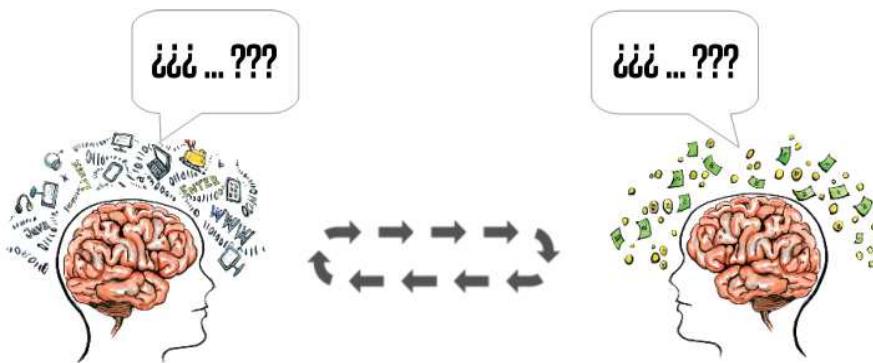
Por otro lado, los desarrolladores se concentran evidentemente en el desarrollo de software. Tal como se muestra aquí, los desarrolladores pueden ser consumidos por los lenguajes de programación y las nuevas tecnologías. Sin embargo, los que trabajan en un proyecto de DDD deben resistirse a la necesidad de estar tan concentrados en lo técnico que no puedan aceptar el foco de la organización en la iniciativa estratégica principal. Mejor dicho, los desarrolladores deben rechazar cualquier sentido y ser capaces de adoptar el *Ubiquitous Language* que gradualmente desarrolla el equipo dentro de su *Bounded Context*.

---

### Concentrarse en la complejidad del negocio, no en las dificultades técnicas

Un usuario utiliza DDD porque la complejidad del modelo de negocio es alta y nunca espera que el modelo de dominio sea más complejo de lo que debiera. Sin embargo, utiliza DDD porque el modelo de negocio es más complejo que los aspectos técnicos del proyecto. ¡Precisamente, por eso los desarrolladores tienen que profundizar en el modelo de negocio con *expertos del dominio*!

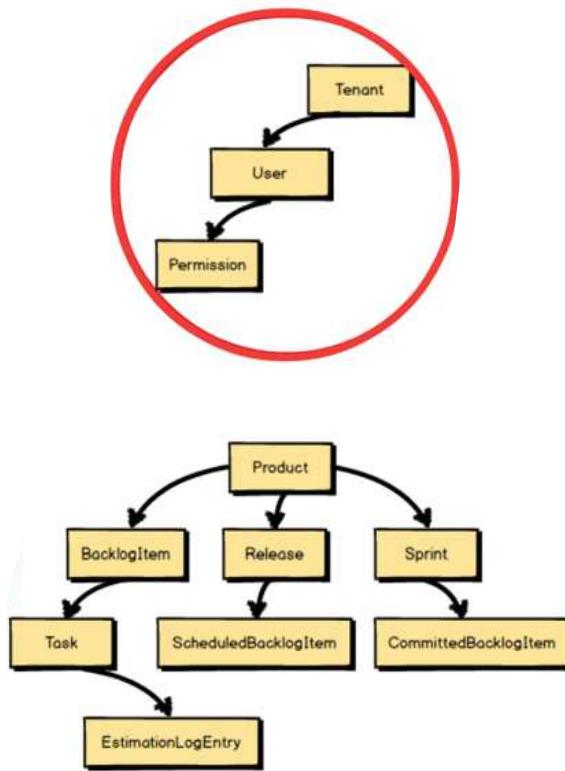
---



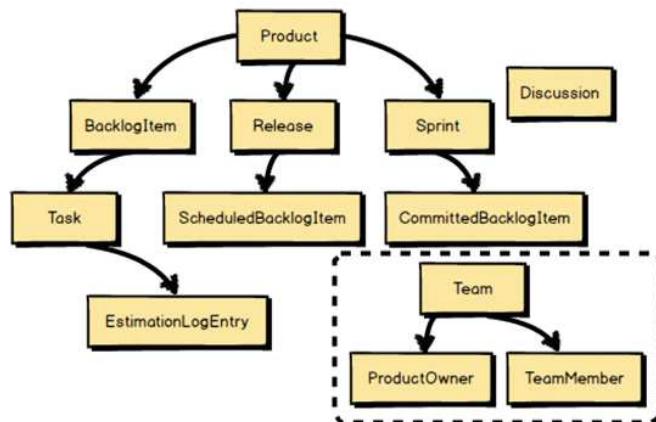
Tanto los desarrolladores como los *expertos de dominio* deben rechazar la tendencia de permitir que los documentos sean más importantes que la conversación. El mejor *Ubiquitous Language* se desarrollará mediante ciclos de retroalimentación colaborativa que dinamicen de forma conjunta el modelo mental común del equipo. La conversación abierta, la exploración y los desafíos a los conocimientos actuales dan como resultado perspectivas más profundas sobre el *Core Domain*.

## Desafiar y unificar

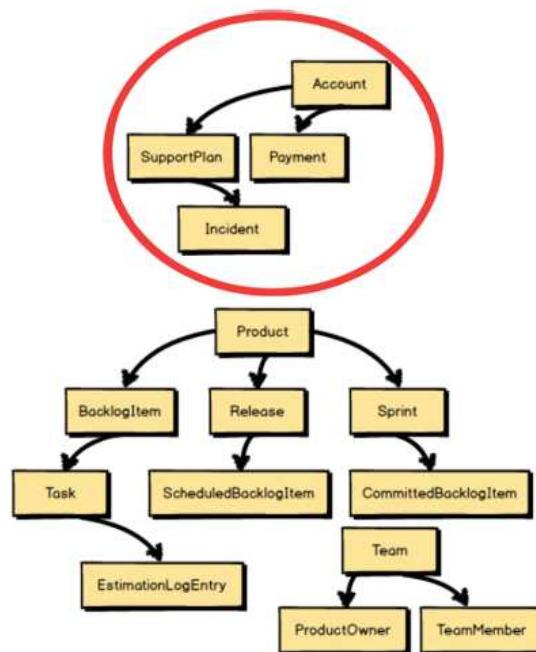
Ahora bien, hay que retomar la pregunta “¿Qué es fundamental?” A partir del modelo del ejemplo anterior que ya estaba fuera de control y en constante expansión, ¡la tarea es desafiar y unificar!



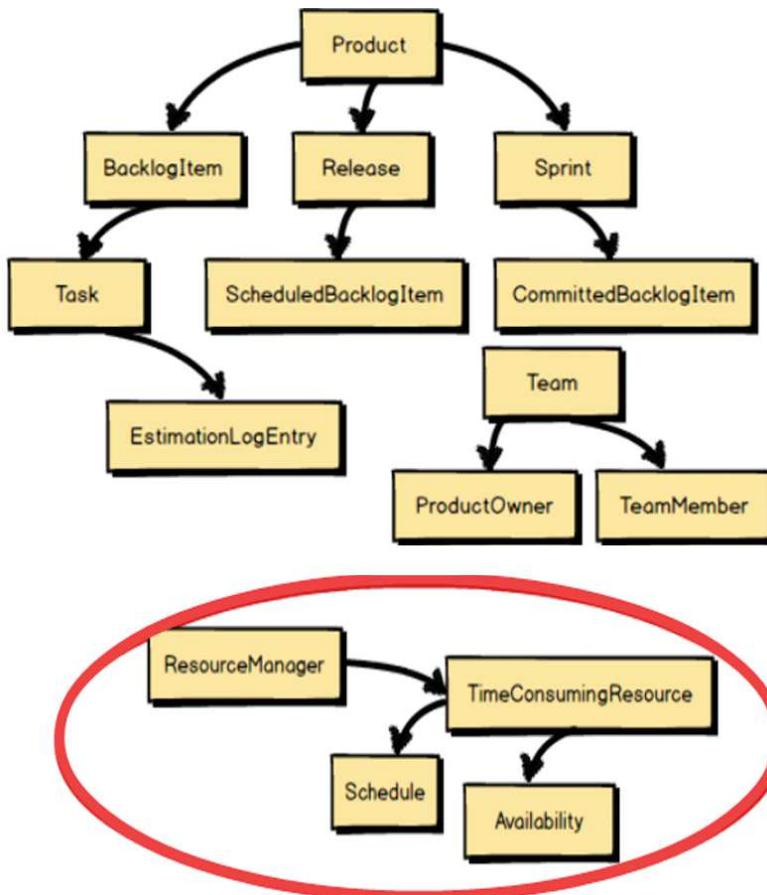
Un desafío sencillo es preguntar si cada uno de los conceptos de este modelo se adhiere o no al *Ubiquitous Language* de Scrum. ¿Cómo comprobar? Por ejemplo, Tenant, User y Permission no tienen nada que ver con Scrum. Estos conceptos deben ser extraídos fuera del modelo de software de Scrum.



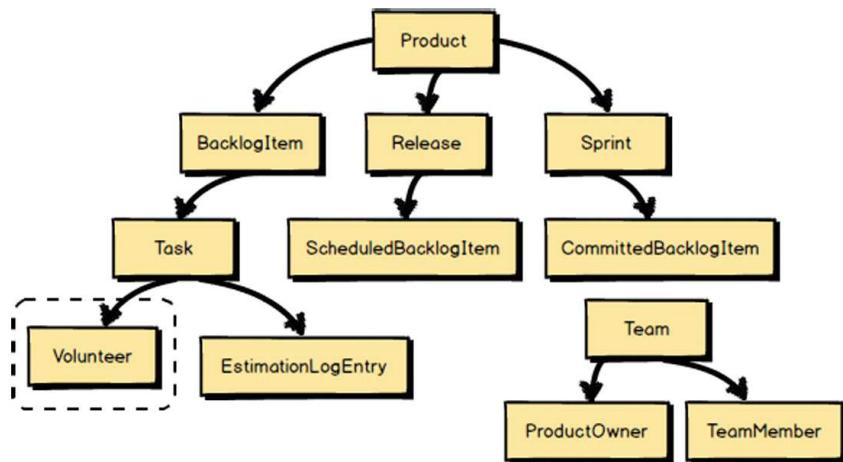
Tenant, User y Permission deben ser remplazados por Team, Product Owner y TeamMember. Un ProductOwner y un TeamMember son en realidad Users en un Tenancy, pero usando los términos ProductOwner y TeamMember, es posible adherirse al *Ubiquitous Language* de Scrum. Por supuesto, son los términos que se utilizan cuando se establecen conversaciones sobre productos basados en Scrum y el trabajo que un equipo realiza con ellos.



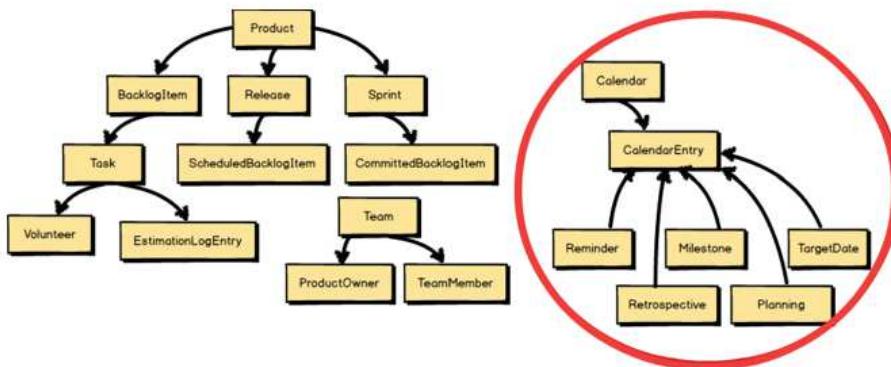
¿Los SupportPlans y los Payments son realmente parte de la gestión de proyectos con Scrum? La respuesta aquí evidentemente es “no”. Es cierto que tanto los SupportPlans como los Payments se administrarán bajo un Tenant’s Account, pero estos no forman parte del lenguaje principal sobre Scrum. Estos términos están fuera de contexto y, por tanto, se eliminan de este modelo.



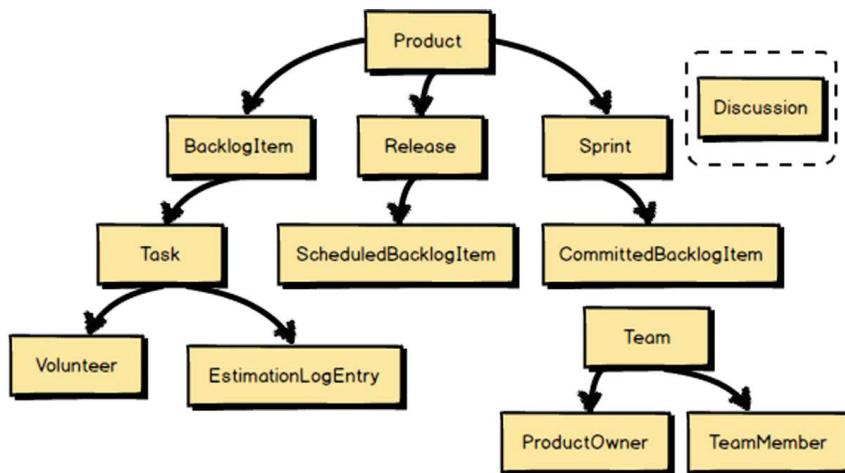
¿Qué pasa entonces con los conceptos relacionados con la *utilización de recursos humanos*? Probablemente sean útiles para alguien, pero no serán utilizados directamente por los **TeamMember** **Volunteers** quienes trabajarán en **BacklogItemTasks**. Está también fuera de contexto.



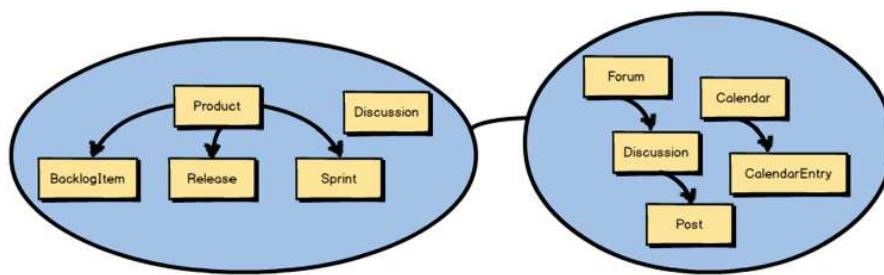
Después de haber añadido Team, ProductOwner y TeamMember, los modeladores se dan cuenta de que les falta un concepto central para permitir que los TeamMembers puedan trabajar en Tasks. En Scrum esto se conoce como Volunteer. Por tanto, el concepto voluntario está en contexto y se debe incluir en el lenguaje del modelo principal.



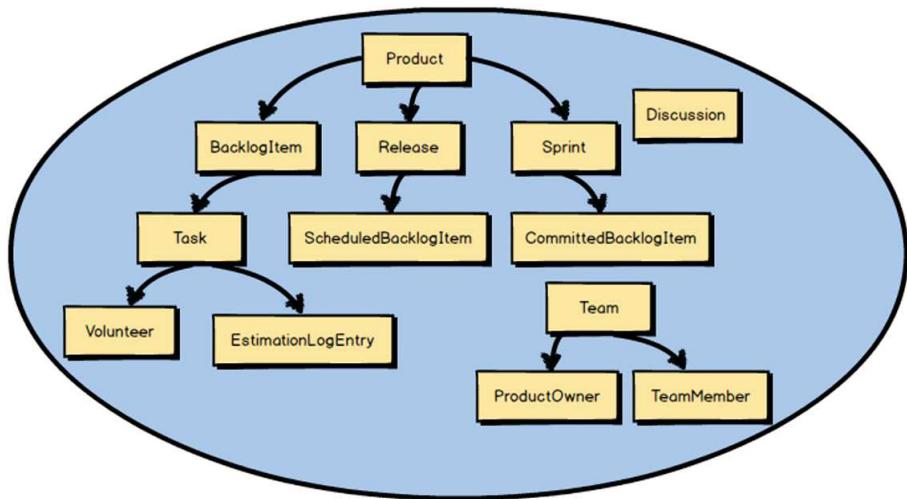
No obstante que los conceptos relacionados con el calendario, como Milestones, Retrospectives y similares están en contexto, el equipo preferiría ahorrar esos esfuerzos de modelamiento para un *sprint* posterior. Están en contexto, pero por ahora están fuera de alcance.



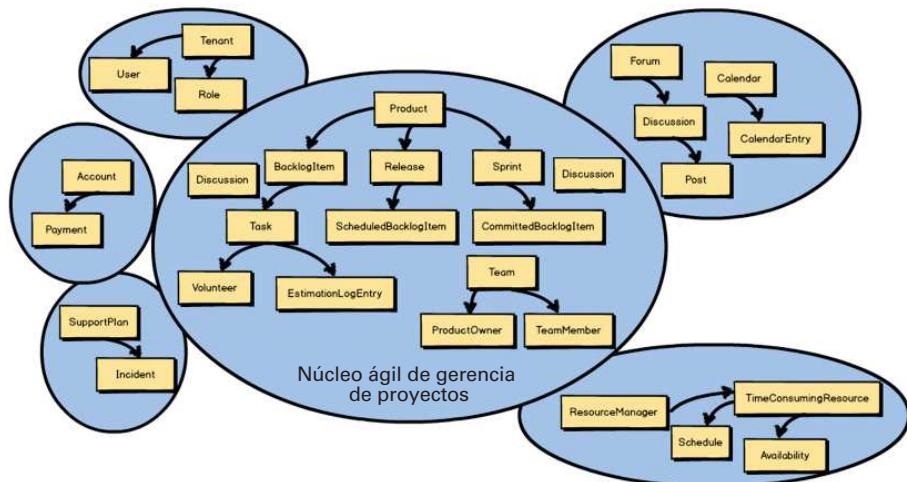
Finalmente, los modeladores quieren asegurarse de que las discusiones formarán parte del *core model*. Así que modelan *Discussions*, y esto significa que esta forma parte del *Ubiquitous Language* del equipo y, por consiguiente, estará dentro del *Bounded Context*.



Estos retos lingüísticos han producido un modelo mucho más limpio y claro de *Ubiquitous Language*. Sin embargo, ¿cómo incluir en el modelo sobre Scrum el concepto de *Discussions* que el equipo necesita? Ciertamente requeriría gran cantidad de componentes de software auxiliares para que funcione, por lo que parece inapropiado modelarlo dentro de nuestro *Bounded Context* de Scrum. De hecho, el paquete completo de colaboración está fuera de contexto. El concepto *discusión* será soportado a través de la integración de otro *Bounded Context*: el *contexto de colaboración*.



Después de este recorrido, queda un *Core Domain* mucho más pequeño. Por supuesto que el *Core Domain* crecerá. Ya se sabe que Planning, Retrospectives, Milestones y los modelos relacionados con el calendario también deben desarrollarse. Aun así, el modelo crecerá solo a medida que nuevos conceptos se adhieran al *Ubiquitous Language* de Scrum.



¿Y qué hay de todos los demás conceptos del modelo que se han eliminado del *Core Domain*? Es muy posible que varios de los demás

conceptos, si no todos, se compongan en sus respectivos *Bounded Contexts*, cada uno de los cuales se adhiere a su propio *Ubiquitous Language*. Más adelante, se verá cómo integrar con ellos mediante el *Context Mapping*.

## Cómo desarrollar un *Ubiquitous Language*

---

Entonces, ¿cómo desarrollar realmente un *Ubiquitous Language* dentro de un equipo con la puesta en práctica de una de las principales herramientas que ofrece DDD? ¿Es posible configurar *Ubiquitous Language* a partir de un conjunto de sustantivos reconocidos? Los sustantivos son importantes, aunque a menudo los desarrolladores de software hacen bastante énfasis en los sustantivos dentro de un modelo de dominio, olvidando que el lenguaje se compone de mucho más que solo nombres. Es cierto que en los ejemplos de *Bounded Contexts*, todo se ha concentrado principalmente en los sustantivos, pero esto eso se debe a que había interés en otro aspecto de DDD, el de reducir el *Core Domain* hasta sus componentes realmente esenciales.

---

### Acelerar el descubrimiento

Para esta parte, es posible probar sesiones de *Event Storming* (fuente de eventos) mientras se trabaja en los escenarios. Esto puede ayudar a comprender rápidamente en qué escenarios debería estar trabajando y cómo priorizar. Del mismo modo, el desarrollo de escenarios concretos dará una mejor idea de la dirección que se debe tomar en las sesiones de *Event Storming*. Son dos herramientas que funcionan bien en conjunto. El uso de *Event Storming* se explica en el capítulo 7 “Herramientas de gestión y aceleración”.

---

No se debe limitar el *Core Domain* a exclusivamente sustantivos; más bien, se debe expresar como un conjunto de escenarios concretos sobre los que se supone se debe construir el modelo de dominio. “Escenarios” no se refiere a casos de uso o historias de usuarios como es común en los proyectos de software. Literalmente, se trata de escenarios en términos de cómo debería funcionar el modelo de dominio, lo que hacen los diversos componentes. Esto se puede lograr de la manera más completa solo colaborando como equipo de *expertos del dominio* y desarrolladores.

A continuación un ejemplo de un escenario que encaja en el *Ubiquitous Language* de Scrum:

*Se ha de permitir que cada ítem del Backlog se pueda comprometer en un sprint. El ítem del Backlog se puede incorporar sólo si ya está planeado para una entrega. Si está comprometido en otro sprint diferente, se tiene que sacar primero. Cuando se compromete en un sprint, se notifica a las partes interesadas.*

Asimismo, hay que tener en cuenta que este no es solo un escenario sobre cómo las personas usan Scrum en un proyecto. No se trata de procesos que llevan a cabo personas; más bien, este escenario es una descripción de cómo se utilizan los componentes reales del modelo de software para apoyar la gestión de un proyecto basado en Scrum.

El escenario anterior no está perfectamente planteado y una ventaja de usar DDD es que constantemente se buscan formas de mejorar el modelo. Sin embargo, este es buen un comienzo. Escuchamos sustantivos, pero el escenario no se limita a sustantivos. También existen verbos y adverbios, y otras formas gramaticales. Asimismo, se escucha que hay restricciones: condiciones que deben cumplirse antes de que el escenario pueda completarse con éxito. El beneficio más importante y la característica más poderosa es que puede haber conversaciones sobre cómo funciona de verdad un modelo de dominio: su diseño.

Incluso es posible hacer imágenes y diagramas sencillos. Se trata de hacer lo que sea necesario para comunicarse de la mejor manera posible dentro el equipo. Sin embargo, también son necesarias unas palabras de advertencia: estar pendiente del tiempo empleado en mantener actualizados documentos de escenarios expresados en palabras, al igual que bocetos y diagramas de un modelo de dominio especialmente a largo plazo. Todo ese material no es el modelo de dominio; más bien, son solo herramientas para ayudar a desarrollar un modelo de dominio. Al final, el código es el modelo y el modelo es el código. Lo protocolario es para celebraciones solemnes, como bodas, no para modelos de dominio. Esto no significa renunciar a ningún esfuerzo por renovar escenarios, sino hacerlo siempre que sea útil y no represente pérdida de tiempo.

¿Qué hacer para mejorar una parte del *Ubiquitous Language* en el ejemplo anterior? ¿Qué falta? En poco tiempo, es probable comprender quién hace el *compromiso* de los ítems del *Backlog* a un *sprint*. Véase a quién hay que agregar y ver qué pasa:

*El product owner añade cada elemento pendiente del backlog al sprint...*

En muchos casos, se debe poner nombre propio a cada rol o persona involucrada en el escenario y asignar un atributo distintivo a otros conceptos, como al ítem del *backlog* o el *sprint*. Esto ayudará a que el escenario sea más concreto y menos como un conjunto de declaraciones sobre criterios de aceptación. Aun así, en este caso particular, no hay buena razón para poner nombre propio al *product owner* o describir con más detalle el ítem del *backlog* o el *sprint*. En esta instancia, todos los *product owners*, todos los ítems de cualquier *backlog* o *sprint* funcionarán de la misma manera, independientemente que tengan o no una persona o identidad concreta. En los casos en que sea útil asignar nombres u otras identidades distintivas a los conceptos del escenario, hay que hacerlo:

*La product owner Isabel, añade al backlog el “Ver perfil de usuario” en el sprint “Entregar perfiles de usuario...”*

Una pausa por ahora. No es que el *product owner* sea el único responsable de decidir que un ítem del *backlog* se compromete en un *sprint*. A los equipos de Scrum no les gustaría mucho, porque estarían comprometidos a entregar software en tiempos en los cuales que no han podido opinar. Aun así, para el modelo de software puede ser más práctico que una sola persona tenga la responsabilidad de llevar a cabo esta acción en el modelo. Entonces, en este caso, se declara que hacer esto es responsabilidad del *product owner*. Incluso así, la naturaleza de los equipos de Scrum obliga a la pregunta “¿Hay algo que deba hacer el resto del equipo para que el *product owner* pueda llevar a cabo el compromiso”?

¿Qué ha pasado? Al desafiar el modelo actual con la pregunta sobre *quién*, se llega a una posibilidad de una perspectiva más profunda del modelo. Quizá se debe requerir al menos consenso de equipo para comprometer un ítem del *backlog* antes de permitir que el *product owner* realice la operación de comprometer un elemento en el *sprint*. Esto podría llevar al siguiente escenario mejorado:

*El product owner agrega un ítem del backlog a un sprint. El ítem del backlog se puede comprometer sólo si el sprint ya tiene fecha de lanzamiento, y si los miembros del equipo han aprobado dicho compromiso por consenso...*

Ahora bien, hay un *Ubiquitous Language* mejorado, porque se ha identificado un nuevo concepto del modelo llamado consenso. Se decidió que debe haber un quórum de miembros del equipo que estén de acuerdo en comprometer un ítem del *backlog*, y debe haber una manera para que aprueben dicho compromiso. Esto introduce un nuevo concepto del modelo y una idea que la interfaz de usuario tendrá que facilitar estas interacciones al equipo. ¿Está clara la idea de cómo se va desarrollando la innovación?

Hay otro punto que falta en el modelo. ¿Cuál? El primer escenario acaba con:

*Cuando el compromiso se complete, se notifica a los interesados.*

¿Cuáles o quiénes son los interesados? Esta pregunta y este reto llevan a más revelaciones durante el proceso de modelado. ¿Quién necesita saber que se ha comprometido un ítem del *backlog* a un *sprint*? De hecho, un elemento importante del modelo es el *sprint* en sí mismo, el cual debe llevar registro del compromiso total y el esfuerzo que se requiere para cumplir todas las tareas del *sprint*. Sin embargo, en este caso se diseña para que el *sprint* lleve este registro. El aspecto importante ahora es que cuando se comprometa un ítem del *backlog*, el *sprint* debe ser notificado:

*Si ya está comprometido en un sprint diferente, primero debe removese de ese sprint. Cuando el compromiso sea efectivo en el nuevo sprint, se debe notificar el sprint origen del que se removió la tarea y también al sprint destino en el que se compromete la tarea ahora.*

Así se presenta un escenario de dominio razonablemente aceptable. Esta frase final también lleva a comprender que el ítem del *backlog* y el *sprint* no necesariamente deben ser conscientes del compromiso al mismo tiempo. Es necesario entonces pedirles a los expertos de negocio que sean conscientes de las consecuencias, aunque este caso parece un claro candidato para introducir consistencia eventual. En el capítulo 5, “Diseño táctico con agregados” se verá por qué es importante y cómo se logra. El escenario mejorado en su totalidad entonces se observa así:

*[...] el product owner compromete un ítem del backlog a un sprint. El ítem del backlog se puede comprometer solo si ya tiene fecha de entrega para su lanzamiento, y si un quórum de parte de los miembros del equipo ha aprobado el compromiso. Si ya está comprometido en un sprint diferente, primero debe removese de ese sprint. Cuando el*

*compromiso sea efectivo en el nuevo sprint, se debe notificar el sprint origen del que se removió la tarea y también al sprint destino en el que se compromete la tarea ahora.*

Entonces, ¿cómo funcionaría un modelo de software en la práctica? Uno se puede imaginar una interfaz de usuario muy innovadora que soporte el modelo de software. Cuando un equipo de Scrum participa en una sesión de planificación de *sprint*, los miembros del equipo usan sus teléfonos inteligentes u otros dispositivos móviles para aprobar o no el compromiso de cada ítem del *backlog* mientras discuten, analizan y acuerdan qué incluir en el próximo *sprint*. Por consenso, los miembros del equipo aprueban cada uno de los ítems del *backlog* y le dan al *product owner* la capacidad de comprometer todos los ítems del *backlog* al *sprint* que hayan sido aprobados previamente.

## Poner en práctica los escenarios

Quizás surja la pregunta de cómo hacer la transición de un escenario escrito hacia un tipo de artefacto que puede usarse para validar un modelo de dominio respecto a las especificaciones del equipo. Para ello se puede utilizar la técnica “*Especificación por ejemplo*”, la cual también se conoce como “*Desarrollo dirigido por comportamiento*” (BDD, por su acrónimo original, *Behavior-Driven Development*). Lo que se intenta lograr con este enfoque es desarrollar y refinar de forma conjunta un *Ubiquitous Language*, modelar con un conocimiento compartido para determinar si el modelo se adhiere a la especificación. La verificación del modelo se realiza mediante la creación de tests de aceptación. Así es como se puede replantear el escenario anterior a una especificación ejecutable:

---

*Escenario: el product owner compromete un ítem del backlog a un sprint.*

*Dado un ítem del backlog que tiene fecha para su lanzamiento.*

*Y el product owner del ítem del backlog.*

*Y un sprint en el cual comprometer.*

*Y un quórum del equipo para aprobar el compromiso.*

*Cuando el product owner compromete el ítem del backlog en el sprint.*

*Entonces, el dueño del Backlog se compromete al sprint.*

*Y se crea el evento sobre el ítem del backlog comprometido.*

---

Al escribir un escenario de esta forma, se puede crear código que lo respalde y usar una herramienta para ejecutar esta especificación. Incluso sin una herramienta, es posible encontrar que esta forma de creación de escenarios con un enfoque *Dado/Cuando/Entonces* funciona mejor que el ejemplo anterior para la creación de escenarios. No obstante, ejecutar las especificaciones como una forma para validar el modelo de dominio será una tentación difícil de resistir. Esto se abordará más adelante en el capítulo 7, “Herramientas de gestión y aceleración”.

Entonces, no es necesario usar esta forma de especificación ejecutable para validar que el modelo de dominio respete los escenarios. Se puede usar un *framework* de tests unitarios para lograr casi lo mismo creando tests de aceptación (no tests unitarios) que validan el modelo de dominio:

---

```
/*
El product owner compromete un ítem del backlog a un
sprint. El ítem del backlog se puede comprometer sólo si
ya tiene fecha de entrega para su lanzamiento, y si un quó-
rum de miembros del equipo ha aprobado el compromiso. Si ya
está comprometido en un sprint diferente, primero debe re-
moverse de ese sprint. Cuando el compromiso sea efectivo en
el nuevo sprint, se debe notificar el sprint origen del que
se removió la tarea y también al sprint destino en el que
se compromete la tarea ahora.
*/



[Test]
public void ShouldCommitBacklogItemToSprint()
{
    // Given
    var backlogItem = BacklogItemScheduledForRelease();
    var productOwner = ProductOwnerOf(backlogItem);
    var sprint = SprintForCommitment();
    var quorum = QuorumOfTeamApproval(backlogItem, sprint);
    // When
    backlogItem.CommitTo(sprint, productOwner, quorum);

    // Then
    Assert.IsTrue(backlogItem.IsCommitted());

    var backlogItemCommitted =
        backlogItem.Events.OfType<BacklogItemCommitted>(). SingleOrDefault();
    Assert.IsNotNull(backlogItemCommitted);
}
```

---

Este enfoque de test de aceptación basado en tests unitarios logra el mismo objetivo que la especificación ejecutable. La ventaja aquí puede ser la capacidad de escribir este tipo de validación de escenario más rápidamente a costa de cierta legibilidad. En cualquier caso, la mayoría de los expertos de dominio deberían seguir este código con la ayuda del desarrollador. Al usar este enfoque probablemente sea buena idea mantener el escenario escrito en los comentarios del código fuente de la prueba, tal y como se ve en el ejemplo.

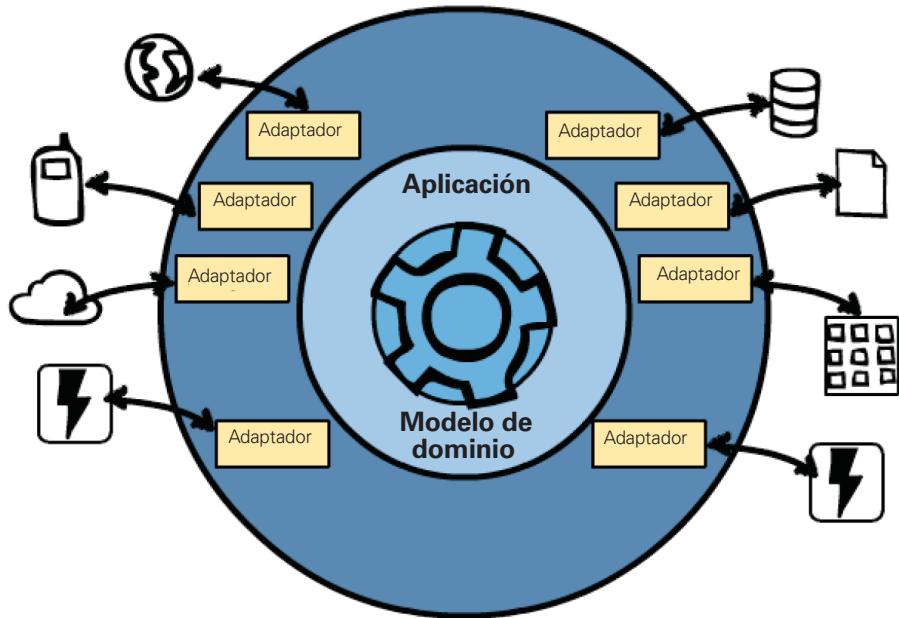
Cualquiera que sea el enfoque escogido, ambos se usarán generalmente al estilo Rojo-Verde o Falla-Pasa en el que la especificación fallará primero cuando se ejecute ya que aún no se ha validado la implementación de los conceptos del modelo de dominio. Paso a paso, se refina el modelo del dominio mediante una serie de resultados en rojo hasta completar totalmente la especificación y pasar la validación (aparecerá todo en verde). Estos tests de aceptación se asocian directamente al *Bounded Context* y se guardarán en el repositorio del código fuente.

## ¿Qué ocurre a largo plazo?

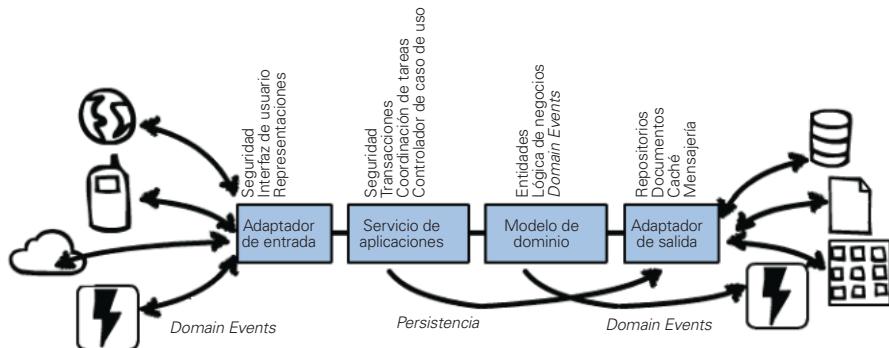
Ahora es posible preguntar cómo mantener el *Ubiquitous Language*, una vez que la innovación ha cesado y llegue el mantenimiento. En realidad, algunos de los mejores aprendizajes o adquisición de conocimientos, tienen lugar durante largos períodos, incluso durante los que algunos podrían referir como “fase de mantenimiento”. Es un error que los equipos consideren que la innovación termina cuando empieza el mantenimiento.

Quizá lo peor que podría pasar es que la etiqueta de “fase de mantenimiento” se adjunte a un *Core Domain*. Un proceso de aprendizaje continuo no es una fase en sí misma. El *Ubiquitous Language* que se desarrolló al inicio debe continuar prosperando a medida que pasan los años. Es cierto que con el tiempo puede ser menos importante, pero probablemente no de forma prolongada. Todo forma parte del compromiso de la organización con las iniciativas clave. Si no se puede hacer este compromiso a largo plazo, ¿este modelo en el que se trabaja hoy constituye un diferenciador estratégico, un *Core Domain*?

## Arquitectura



Quizá surja otra pregunta: ¿Qué hay en un *Bounded Context*? Al usar este diagrama de arquitectura de *puertos y adaptadores* (IDDD), es posible observar que un *Bounded Context* se compone de más de un modelo de dominio.



Estas capas son comunes en un *Bounded Context*: *adaptadores de entrada*, como los controladores de la interfaz de usuario, controladores API tipo REST y consumidores de mensajes, *servicios de aplicación* que organizan casos de uso y gestionan transacciones, el modelo de dominio en el que nos hemos concentrado y *adaptadores de salida*, tales como la gestión de persistencia y los productores de mensajes. Hay mucho que decir acerca de las diversas capas en esta arquitectura y son demasiados detalles para trabajar en este libro; por ello, consulte el capítulo 4 de *Implementing Domain-Driven Design* [IDDD] para un análisis exhaustivo.

---

## Modelo de dominio libre de tecnología

Aunque hay tecnología dispersa por toda la arquitectura, el modelo de dominio debe estar libre de ella. Por ejemplo, ese es el motivo por el que las transacciones son gestionadas por los servicios de aplicación y no por el modelo de dominio.

---

Puertos y adaptadores se pueden usar como arquitectura de base, aunque no es la única alternativa para utilizar junto con DDD. A partir de puertos y adaptadores, se puede usar DDD con cualquiera de las siguientes arquitecturas o patrones arquitecturales, mezclándolos y combinándolos según las necesidades:

- Arquitectura basada en eventos (*Event-Driven Architecture*); *Event Sourcing* [IDDD]. Téngase en cuenta que en este libro en el capítulo 6 se analizará la *Event Sourcing*, “Diseño táctico con *Domain Events*”.
- CQRS: segregación de responsabilidades entre comandos y consultas (*Command Query Responsibility Segregation* [IDDD]).
- Reactividad y modelo de actores; véase *Reactive Messaging Patterns with the Actor Model* [Reactive] que también profundiza en el uso del modelo de actores con DDD.
- REST: transferencia del estado representacional (Representational State Transfer) [IDDD].
- SOA: arquitectura orientada a servicios (Service-Oriented Architecture) [IDDD].
- Microservicios que se explican en *Building Microservices* [microservicios], fundamentalmente equivalentes a *Bounded Contexts* de DDD, así que tanto este libro como *Implementing Domain-Driven*

*Design* [IDDD] abordan el desarrollo de microservicios desde esa perspectiva.

- Computación en la nube se incluye de la misma forma que los microservicios. Así que todo lo que aparece en este libro, al igual que en *Implementing Domain-Driven Design* [IDDD] y en *Reactive Messaging Patterns with the Actor Model* [Reactive] es aplicable.

Otro comentario pertinente sobre los microservicios: algunos consideran que un microservicio es mucho más pequeño que un *Bounded Context* de DDD. Si se asume esa definición, un microservicio modelaría un único concepto del dominio y gestionaría un tipo de datos muy pequeño. Un ejemplo de ese tipo de microservicio sería un *Product* y otro es un *BacklogItem*. Si esta es la granularidad que se considera adecuada para un microservicio, se entiende que tanto el microservicio del *Product* como el microservicio del *BacklogItem* pertenecerán igualmente a un mismo *Bounded Context* lógico más grande. Esos dos componentes más pequeños en forma de microservicio tienen únicamente despliegues independientes, lo que también puede tener impacto en la forma como interactúan (véase *Context Mapping–mapeo entre contextos*). Lingüísticamente, aún se encuentran en los mismos límites contextuales y semánticos basados en Scrum.

## Resumen

---

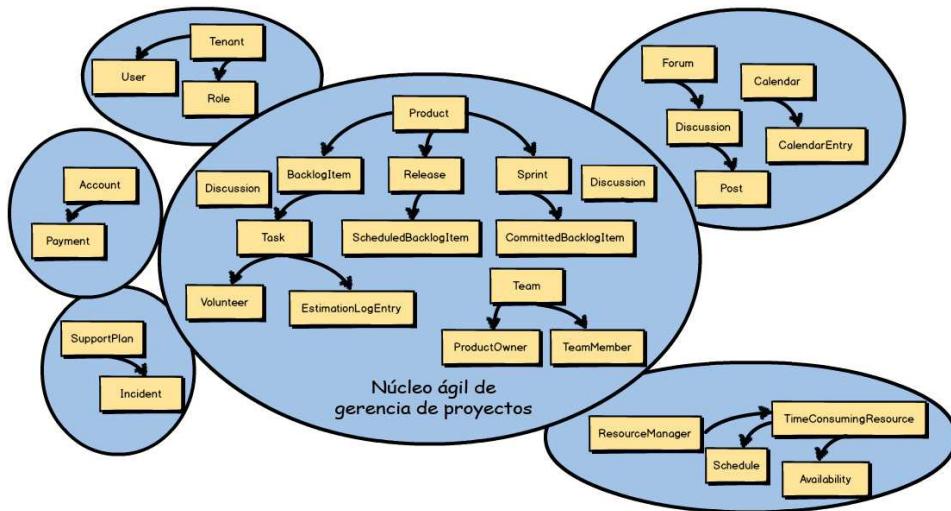
En síntesis, lo aprendido:

- Algunos de los principales riesgos de poner demasiadas cosas en un mismo modelo y producir una *Big Ball of Mud*.
- La aplicación del diseño estratégico en DDD.
- El uso de *Bounded Context* y *Ubiquitous Language*.
- Cómo desafiar supuestos y unificar modelos mentales.
- Cómo desarrollar un *Ubiquitous Language*.
- Diferentes componentes arquitecturales hallados dentro de un *Bounded Context*.
- ¡Que DDD no es muy difícil de poner en práctica!

Para un tratamiento más profundo de los *Bounded Contexts*, véase capítulo 2 de *Implementing Domain-Driven Design* [IDDD].

# CAPÍTULO 3

## Diseño estratégico con subdominios



Cuando se trabaja en un proyecto con DDD, siempre hay múltiples *Bounded Contexts* en juego. Uno de los *Bounded Contexts* será el *Core Domain*, y además también habrá varios *subdominios* en otros *Bounded Contexts*. En el capítulo anterior, se observó la importancia de dividir diferentes modelos por su *Ubiquitous Language* específico y formar múltiples *Bounded Contexts*. En el diagrama anterior, hay seis *Bounded Contexts* y seis *subdominios*. Debido a que se utilizó el diseño estratégico de DDD, los equipos lograron el resultado más óptimo posible para un modelo: un *subdominio* por cada *Bounded Context*, y un *Bounded Context* por cada *subdominio*. En otras palabras, el núcleo del administrador ágil de proyectos es a la vez un *Bounded Context* y un *subdominio*. En algunas situaciones, puede haber varios *subdominios* en un *Bounded Context*, pero eso no logra el resultado óptimo.

## ¿Qué es un subdominio?

---

En pocas palabras, un *subdominio* es una subparte del dominio general del negocio. Se puede pensar entonces en un subdominio como una representación única y lógica del modelo de dominio; la mayoría de los dominios de un negocio suelen ser demasiado grandes y complejos para concebirlos como un todo, por lo que generalmente, solo hay preocupación de los subdominios que se deben usar en determinado proyecto. Los subdominios se pueden usar para dividir lógicamente el dominio de negocio, para comprender el *espacio* del problema en un proyecto grande y complejo.

Otra forma de pensar en un subdominio es como un área bien definida sobre el conocimiento del negocio, suponiendo que esa área es responsable de proporcionar una solución a una unidad clave del negocio. Esto implica que el subdominio en particular tendrá uno o más expertos del dominio que entienden muy bien los aspectos del negocio, el cual facilita ese subdominio. El subdominio también puede tener mayor o menor importancia estratégica para el negocio.

Si se hubiera utilizado DDD para desarrollarlo, el subdominio se habría implementado como un *Bounded Context*. Los expertos del dominio, especializados en esa área en particular del negocio, habrían sido miembros del equipo que desarrolló el *Bounded Context*. Si bien el uso de DDD para desarrollar un *Bounded Context* limpio es la alternativa óptima, en ocasiones solo es posible desear que así se hubiera hecho.

## Tipos de subdominios

---

Existen tres tipos principales de subdominios en un proyecto:

- *Core Domain*: en este punto es donde se realiza una inversión estratégica en un modelo de dominio único y bien definido, lo que compromete recursos significativos para construir cuidadosamente el *Ubiquitous Language* en un *Bounded Context* explícito. Este proyecto debe tener la máxima prioridad en la organización, porque así se distinguirá de todos los competidores. Ya que la organización no se puede destacar en todo lo que hace, el *Core Domain* subraya dónde debe ser excelente. Alcanzar el nivel de aprendi-

zaje y comprensión necesarios para tomar una decisión de este tipo requiere compromiso, colaboración y experimentación. Es ahí donde la organización necesita invertir más deliberadamente en software. Los medios para acelerar y administrar dichos proyectos de manera eficiente y efectiva aparecen más adelante en este libro.

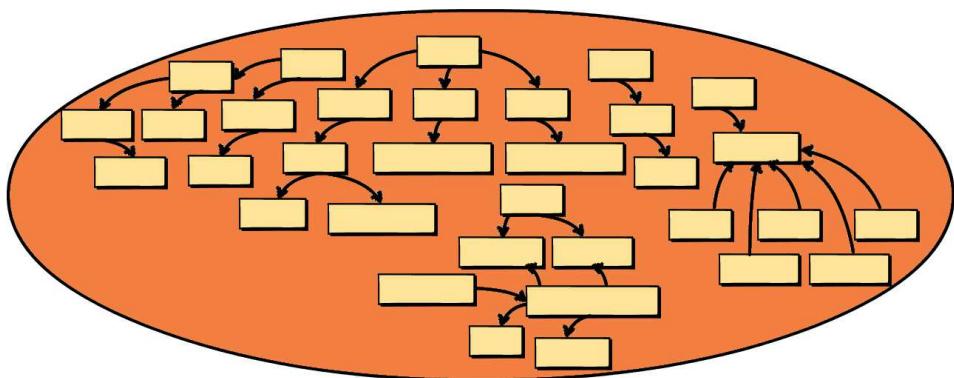
- Subdominio de soporte: esta es una de esas situaciones al modelar que requiere un desarrollo a la medida y personalizado porque no existe ninguna solución en el mercado que se pueda adquirir. Aun así, no se invertirá tanto como en el *Core Domain*. Entonces, se ha de contemplar la posibilidad de subcontratar este tipo de *Bounded Context* para así evitar confundirlo con algo que distinga estratégicamente al negocio, e invertir demasiado en él. Sigue siendo, sin embargo, una pieza del modelo de software importante, ya que el *Core Domain* no podría ser exitoso sin él.
- Subdominio genérico: este tipo de solución puede estar ya disponible en el mercado, aunque también puede ser subcontratado o desarrollado internamente por un equipo que no tenga el tipo de desarrolladores de élite que se asignaría al *Core Domain* o incluso a un dominio de soporte. Por ello, se debe tener cuidado de no confundir un subdominio genérico con un *Core Domain*. No es aconsejable hacer ese tipo de inversión aquí.

Cuando se habla de un proyecto en el que se utiliza DDD, lo más probable es que se esté analizando un *Core Domain*.

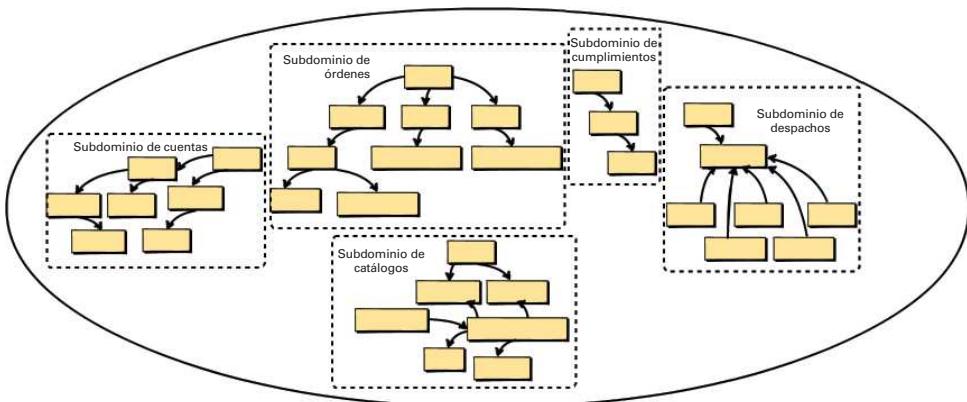
## Cómo confrontar la complejidad

---

Es posible que algunos de los actuales límites del sistema que tiene un dominio de negocio sean sistemas legados, probablemente creados directamente por la organización o mediante la adquisición de licencias de software. Es posible que no se pueda hacer mucho para mejorar esos sistemas legados, pero aún se deben considerar cuando tengan impacto en el proyecto del *Core Domain*. Para hacerlo, se usan los *subdominios* como herramienta para discutir el *espacio del problema*.



Infortunadamente, pero no por ello menos cierto, algunos sistemas legados son tan contrarios a la forma DDD de diseñar con *Bounded Contexts* que incluso podría referirse a ellos como sistemas legados sin límites. Un sistema legado de esas características es lo que ya se ha denominado una *Big Ball of Mud*. En otras palabras, ese único sistema está lleno de múltiples modelos tan enmarañados que debieron haber sido diseñados e implementados por separado, pero se fusionaron en un embrollo muy complejo e interrelacionado.

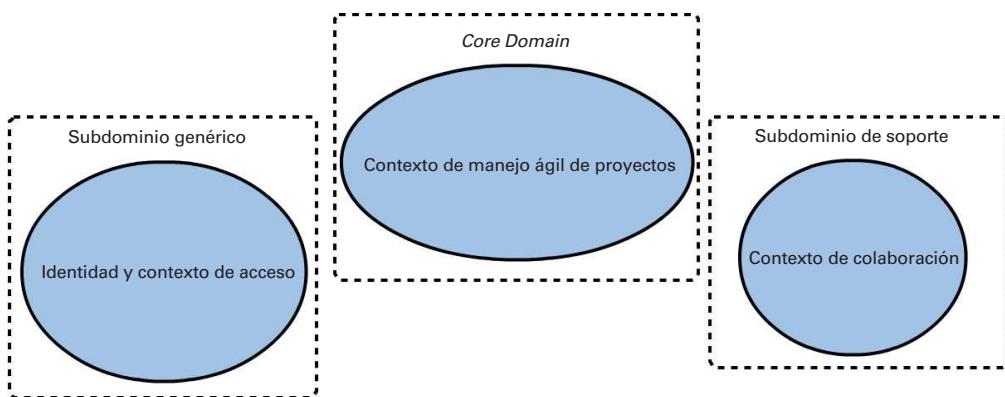


Dicho de otra forma, cuando se analiza un sistema legado, pueden coexistir unos pocos o incluso muchos modelos de dominio *lógicos* dentro de ese mismo sistema legado. Entonces, se debe pensar en cada uno

de esos modelos de dominio lógicos como un subdominio diferente. En la figura, cada subdominio lógico dentro de esa *Big Ball of Mud* monolítica legada está marcado por un cuadro discontinuo. Hay cinco modelos lógicos o subdominios. Tratar los subdominios lógicos como tal ayuda a lidiar con la complejidad de grandes sistemas. Esto tiene sentido, porque permite tratar el espacio del problema como si se hubiera desarrollado utilizando DDD y múltiples *Bounded Contexts*.

El sistema legado parece menos monolítico y lodoso si se imaginan distintos lenguajes ubicuos, al menos para comprender cómo es posible integrarse con este. Pensar y analizar tales sistemas legados mediante subdominios ayuda a enfrentar la dura realidad de un gran modelo que está enmarañado. Y mientras se usa esta herramienta es posible diferenciar los subdominios más valiosos y necesarios para el negocio y para el proyecto en mente, de aquellos que pueden ser relegados a un estado inferior.

Con esto en perspectiva, es posible incluso mostrar el *Core Domain* en el que se trabaja o en el que se vaya a desarrollar en la misma figura. Esto ayudará a comprender relaciones y dependencias entre subdominios. Los detalles sobre ese tema se abordarán en el capítulo sobre *Context Mapping*.



Cuando se usa DDD, un *Bounded Context* debe alinearse uno a uno (1:1) con un solo subdominio. Es decir, cuando se usa DDD, si hay un único *Bounded Context*, debe haber también como objetivo, un único modelo de ese subdominio en ese *Bounded Context*. No siempre será

posible o práctico lograrlo, pero cuando sea posible, es importante diseñar de esa manera. Eso mantendrá los *Bounded Contexts* limpios y enfocados en la iniciativa estratégica principal.

Si se ha de crear un segundo modelo en el mismo *Bounded Context* (dentro del *Core Domain*), se debe separar el modelo secundario del *Core Domain* utilizando un módulo completamente separado [IDDD]. (Un módulo DDD es esencialmente un paquete en Scala y Java, y un *name-space* en F# y C#). Con esto se consigue una evidente declaración lingüística que uno es el modelo central y el otro es simplemente un modelo de soporte. Esta práctica de segregar un subdominio se aplicará en el espacio de la solución.

## Resumen

---

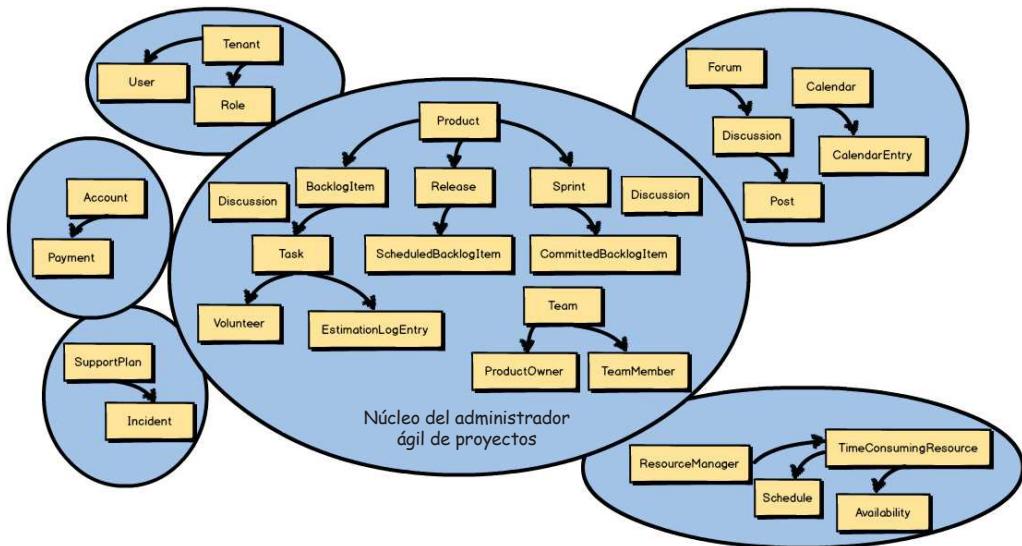
En síntesis, lo aprendido:

- Qué son los subdominios y cómo se usan, tanto en el espacio del problema como en el ámbito de la solución.
- La diferencia entre un *Core Domain*, un subdominio de soporte y un subdominio genérico.
- Cómo se puede hacer uso de los subdominios mientras se plantea la integración con un sistema legado tipo *Big Ball of Mud*.
- La importancia de alinear un solo *Bounded Context* DDD con un solo subdominio.
- Cómo se debe segregar un modelo de subdominio de soporte del modelo de *Core Domain* mediante un módulo DDD cuando no sea práctico separar los dos en diferentes *Bounded Contexts*.

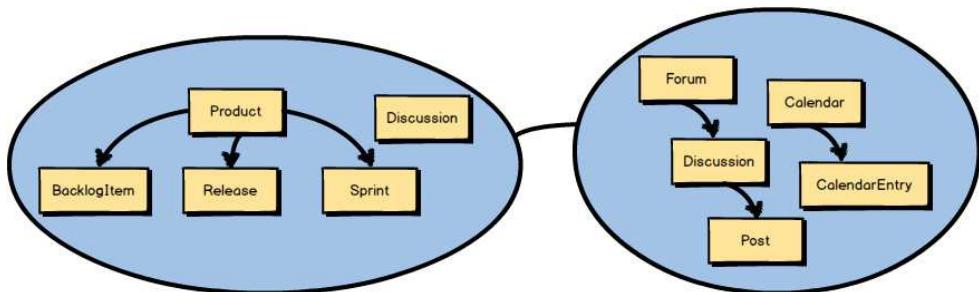
Para mayor información sobre subdominios, véase el capítulo 2 de *Implementing Domain-Driven Design* [IDDD].

# CAPÍTULO 4

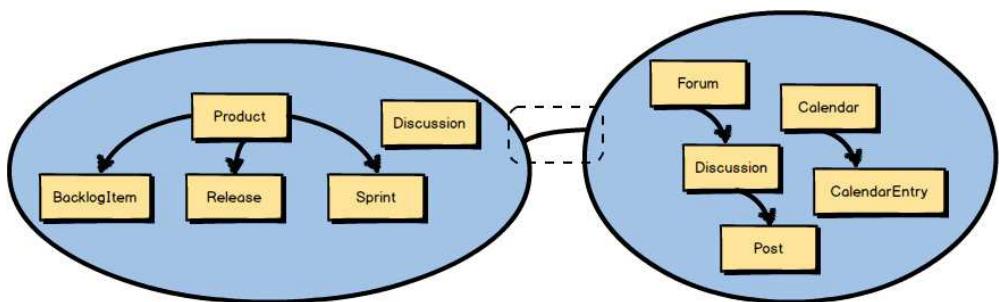
## Diseño estratégico con *Context Mapping*



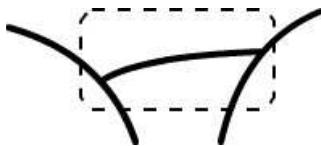
En los capítulos anteriores se abordó y se reconoció asimismo que, además del *dominio principal*, existen múltiples *Bounded Contexts* asociados a cada proyecto DDD. Todos los conceptos que no pertenecían al *contexto de núcleo del administrador ágil de proyectos*, el objeto de trabajo del *dominio principal*, se trasladaron a otro de los *Bounded Contexts*.



También se reconoció que el *dominio principal* sobre *núcleo del administrador ágil de proyectos*, tiene que integrarse con otros *Bounded Contexts*. Esta integración se conoce en DDD como *Context Mapping*, tal como se puede observar en el diagrama en el anterior donde *discusión* existe en ambos *Bounded Contexts*. Recuérdese que esto se debe a que el *contexto de colaboración* es la fuente de discusión y que el *contexto del núcleo del administrador ágil de proyectos*, es el consumidor de *discusión*.



El *Context Mapping* se representa en este diagrama con la línea continua dentro del cuadro discontinuo. El cuadro discontinuo no forma parte del *Context Mapping*, sólo se usa para resaltar la línea. En realidad, esta línea continua entre los dos *Bounded Contexts* es lo que representa un *Context Mapping*. En otras palabras, la línea indica que los dos *Bounded Contexts* están relacionados de alguna manera. Por consiguiente, existe dinámica entre los equipos de los dos *Bounded Contexts*, así como cierta integración.



Teniendo en cuenta que en dos *Bounded Contexts* conviven dos *lenguajes ubicuos* diferentes, esta línea representa la traducción entre esos dos idiomas. A modo ilustrativo, supóngase que dos equipos necesitan trabajar juntos pero se encuentran en países diferentes y no hablan el mismo idioma. Estos equipos necesitan un intérprete, o alternativamente, uno o ambos equipos deberían aprender mucho sobre el idioma del otro. Encontrar un intérprete resultaría menos trabajoso para ambos equipos pero podría ser más costoso si se consideran otros aspectos. Por ejemplo, el tiempo adicional necesario para que un equipo hable con el intérprete para que luego éste transmita esas declaraciones al otro grupo. Podría funcionar bien durante unos primeros momentos pero luego se volvería engorroso. Aun así, los equipos podrían encontrar que esta solución es mejor que tener que aprender un idioma extranjero y cambiar constantemente entre lenguas. Y, por supuesto, esto describe la relación sólo entre dos equipos. ¿Qué pasa si hay más equipos involucrados? Los mismos pros y contras aplican al traducir de un *Ubiquitous Language* a otro, o alternativamente, al adaptarse a otro *Ubiquitous Language*.

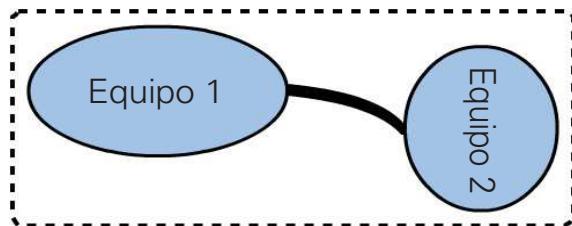


Cuando se habla de *Context Mapping*, lo que interesa es *qué tipo* de relación e integración entre equipos está representada por la línea entre cualesquier dos *Bounded Contexts*. Establecer límites y contratos entre éstos ayuda a controlar los cambios que se introducen durante el tiempo. Existen varios tipos de *Context Mapping*, tanto técnicos como de personas y equipos, que pueden representarse con esa línea. En algunos casos, ambas relaciones, la técnica y la de equipos, serán la misma o serán afectadas por los mismos criterios.

## Tipos de mapeo

¿Qué relaciones e integraciones se pueden representar por la línea del *Context Mapping*? Obsérvese a continuación.

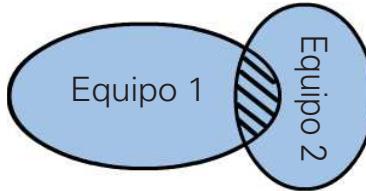
### Alianza



Una relación de *alianza* existe entre dos equipos. Cada equipo es responsable de un *Bounded Context*, y éstos crean una coalición para alinear los dos equipos a un conjunto de objetivos dependientes. Se dice entonces que los dos equipos tendrán éxito o fracasarán juntos. Al estar tan alineados, éstos se reunirán con frecuencia para sincronizar horarios y trabajo dependiente, y tendrán que usar integración continua para mantener sus integraciones en armonía. La sincronización se representa por la línea gruesa de mapeo entre los dos grupos. La línea gruesa indica el nivel de compromiso requerido que es bastante alto.

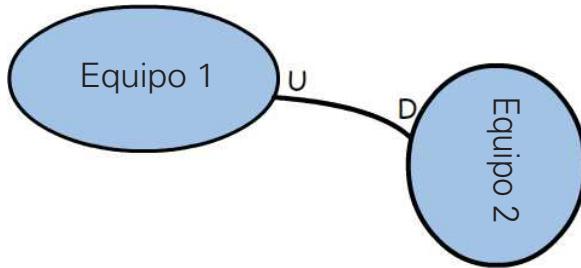
Possiblemente es difícil mantener una *alianza* a largo plazo, por lo que los equipos que forman una alianza deben hacer lo posible para establecer límites sobre el término de su relación. La *alianza* debe durar sólo mientras constituya una ventaja y debe ser reasignada a una relación diferente cuando la ventaja se agote por pérdida o agotamiento del compromiso.

## Kernel compartido



Un *kernel compartido*, representado en el diagrama anterior por la intersección de otros dos *Bounded Contexts*, describe la relación entre dos o más equipos que comparten un modelo pequeño aunque común. Los equipos deben ponerse de acuerdo sobre qué elementos del modelo van a compartir. Es posible que sólo uno de los equipos mantenga el código, compile y pruebe lo que se comparte. Un *kernel compartido* a menudo es muy difícil de concebir y de mantener, porque debe haber comunicación abierta entre los equipos y acuerdo constante sobre lo que constituye el modelo por compartir. Aun así, es posible tener éxito si todos los involucrados están comprometidos con la idea que el kernel es mejor que ir por *caminos separados*, tema por explicar más adelante.

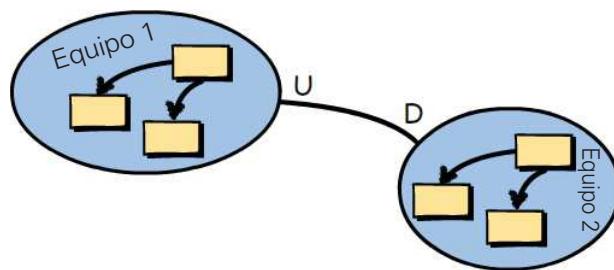
## Relación cliente-proveedor



Un *cliente-proveedor* describe la relación entre dos *Bounded Contexts* y los equipos respectivos, donde el *proveedor* está en la *upstream* (zona alta) de la dependencia, marcado con una U en el diagrama, y el *cliente* está en la *downstream* (zona baja) marcado con una D en el diagrama. El *proveedor* domina esta relación porque debe proporcionar lo que el *cliente* necesita. Depende del *cliente* planificar con el *proveedor* para cumplir varias expectativas, aunque al final, el *proveedor* determina lo que recibirá el *cliente* y cuándo. Ésta es una relación muy común y práctica.

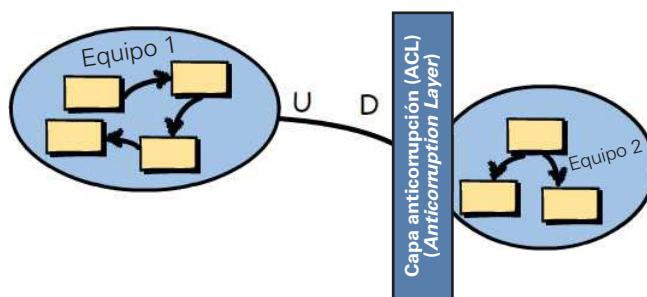
tica entre los equipos, incluso dentro de la misma organización, siempre y cuando la cultura corporativa no permita al *proveedor* ser completamente autónomo y no responder a las necesidades reales de los *clientes*.

## Relación conformista



Existe una relación *conformista* cuando hay equipos *upstream* y *downstream*, y el equipo *upstream* no tiene ninguna motivación para apoyar las necesidades específicas del equipo *downstream*. Por diversas razones, el equipo *downstream* no puede sostener un esfuerzo para traducir el *Ubiquitous Language* del modelo *upstream* para que se ajuste a sus necesidades específicas y se acaba ajustando al modelo *upstream* tal como está. Un equipo a menudo se convierte en *conformista*, por ejemplo, en una integración con un modelo muy grande y complejo que ya está muy establecido; por ejemplo, la necesidad de ajustarse al modelo de Amazon.com cuando un usuario se integra como uno de los vendedores afiliados de este sitio web.

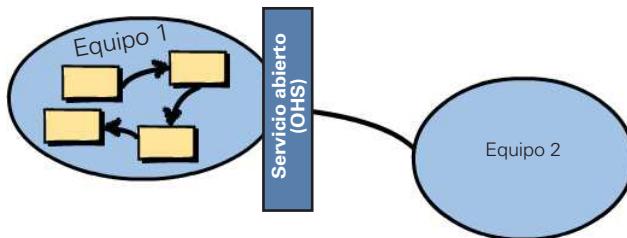
## Capa anticorrupción (*Anticorruption Layer*)



Una capa anticorrupción (*Anticorruption Layer*) es la relación más defensiva en el *Context Mapping*: el equipo *downstream* crea una capa de traducción entre su *Ubiquitous Language* (modelo) y el *Ubiquitous Language* (modelo) que está *upstream*. La capa aísla el modelo *downstream* del modelo *upstream* y traduce entre los dos; por consiguiente, éste también es un enfoque para la integración.

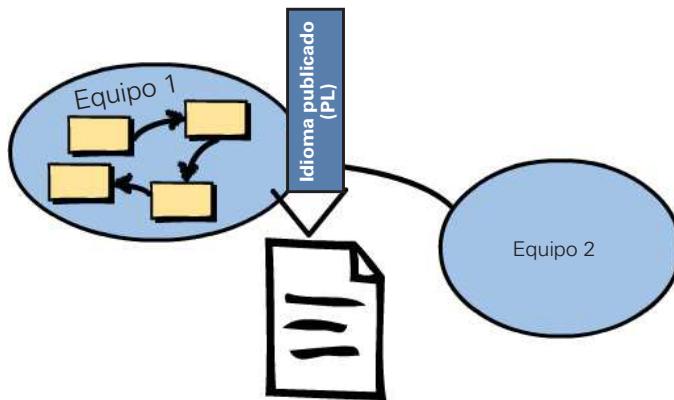
Siempre que sea posible, se debe intentar la creación de una *capa anticorrupción* entre el modelo *downstream* y el modelo de integración *upstream*, de tal modo que se puedan producir conceptos de modelo en el lado de la integración que se ajusten específicamente a las necesidades empresariales y mantenerse completamente aislado de conceptos externos. Sin embargo, al igual que contratar a un traductor para que actúe entre dos equipos que hablan diferentes idiomas, el costo podría ser demasiado alto en algunos casos.

## Servicio abierto



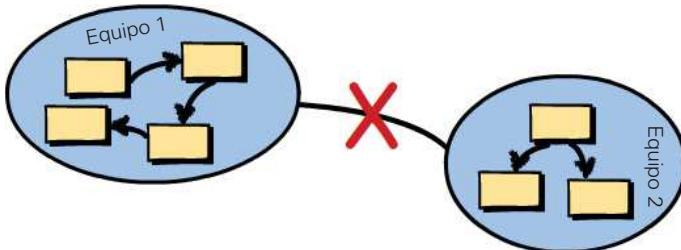
Un servicio abierto (*Open Host Service*) define un protocolo o interfaz que da acceso al *Bounded Context* mediante un conjunto de servicios. El protocolo es “abierto” para que todos los que necesiten integrar con el *Bounded Context*, puedan usarlo con relativa facilidad. Los servicios ofrecidos por la interfaz de programación de aplicaciones (API) están debidamente documentados y son agradables de usar. Incluso si alguien pertenece al equipo 2 en este diagrama y no pudiera tomarse el tiempo para crear una capa anticorrupción aislante para el lado de la integración, sería mucho más tolerable ser *conformista* con este modelo que con muchos sistemas legados que se pudieran encontrar. Entonces, se podría decir que el lenguaje del servicio abierto es mucho más fácil de consumir que el de otros tipos de sistemas.

## Idioma publicado (*Published Language*)



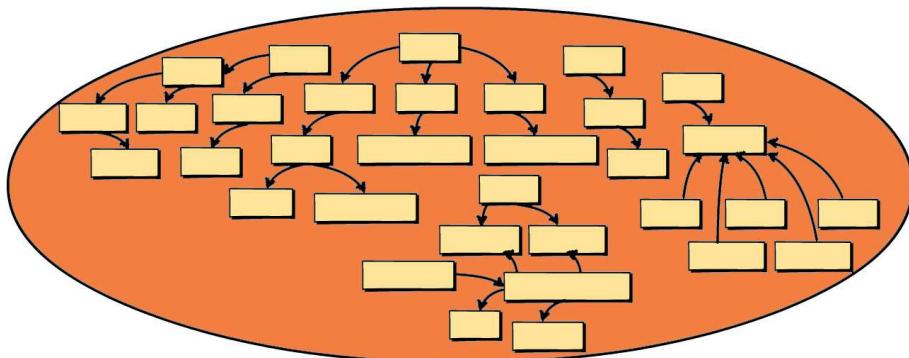
Un idioma publicado es un lenguaje de intercambio de información debidamente documentado que posibilita consumo sencillo y traducción de varios *Bounded Contexts*. Los consumidores que leen y escriben pueden traducir desde y hacia el idioma compartido con la confianza que sus integraciones son correctas. Dicho idioma publicado se puede definir con XML Schema, JSON Schema o un formato de conexión más óptimo, como Protobuf o Avro. A menudo, un *Open Host Service* sirve y consume un idioma publicado proporcionando la mejor experiencia de integración para terceros. Esta combinación simplifica enormemente la traducción entre dos *lenguajes ubicuos*.

## Caminos separados (*Separate Ways*)

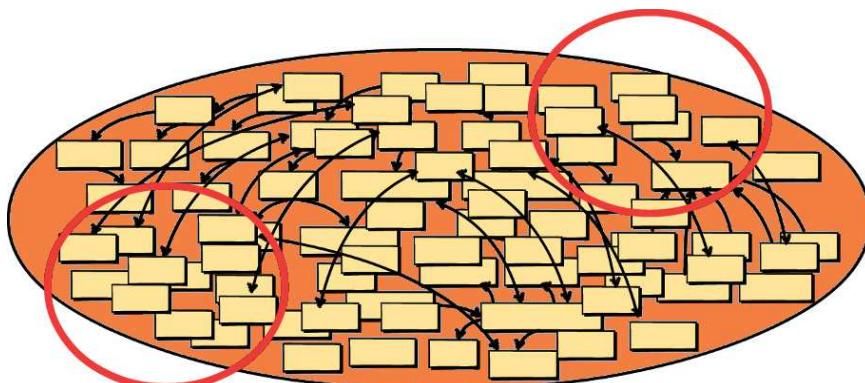


Caminos separados describe una situación en la que la integración con uno o más *Bounded Contexts* no producirá beneficios significativos mediante consumir varios *Ubiquitous Language*. Quizás la funcionalidad que busca no está proveída por ningún *Ubiquitous Language*. En este sentido, implemente su propia solución especializada en su *Bounded Context* y olvide la integración, en este caso particular.

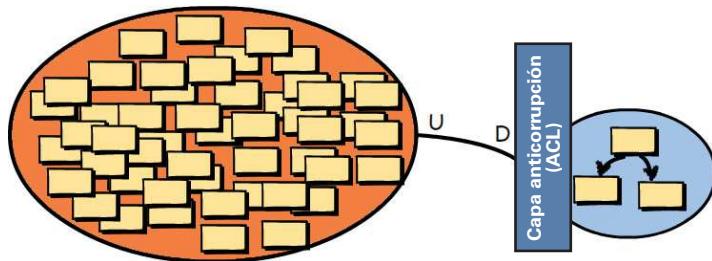
### *Big Ball of Mud*



En los capítulos anteriores se abordó en gran medida el tema de la *Big Ball of Mud*, aunque es importante reforzar problemas graves que se pueden experimentar cuando haya que trabajar o integrarse a una de ellas. Crear una *bola de lodo* de esta naturaleza debe evitarse a toda costa.



Por experiencia, esto es lo que sucede con el paso del tiempo cuando se crea una *Big Ball of Mud*: (1) el número de *agregados* va creciendo y se contaminan entre sí debido a conexiones y dependencias injustificadas. (2) mantener una parte de la *Big Ball of Mud* causa incertidumbres por todo el modelo, lo que conduce a estar apagando incendios sin resultados concretos. (3) sólo el conocimiento tribal y los héroes que hablan todos los idiomas a la vez pueden salvar el sistema de un gran colapso.



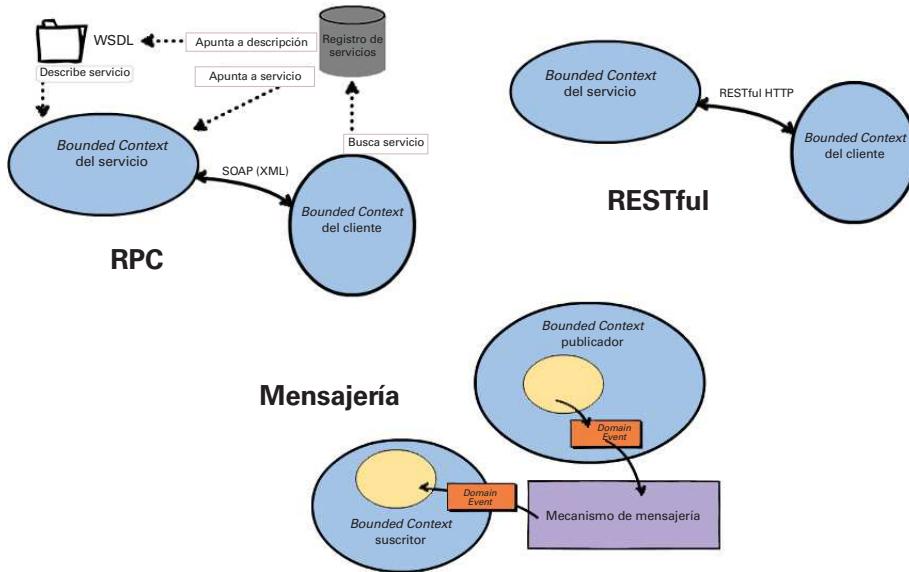
El problema es que ya existen muchas *bolas grandes de lodo* en el mundo de los sistemas de software, y sin duda, el número aumenta cada vez más. Incluso aunque se puede evitar una *Big Ball of Mud*, gracias a las técnicas de DDD, es posible que haya necesidad de integrarse a más de una. Pero si así fuera, intentar crear una capa anticorrupción que proteja cada sistema legado del modelo propio de ese inteligible laberinto pues, si no se hace, contaminaría el modelo que se pretende construir. Por ello, independientemente de lo que se haga, ¡no hablar ese idioma!

## Cómo hacer uso apropiado del *Context Mapping*

Possiblemente surge el interrogante sobre qué tipo específico de interfaz se debería suministrar para permitir integrarse a determinado *Bounded Context*. Eso depende de lo que proporcione el equipo que sea dueño de dicho *Bounded Context*. Podría ser RPC vía SOAP, o interfaces RESTful con recursos, o podría ser una interfaz de mensajería usando colas o un modelo publicar-subscribir (*Publish-Subscribe*). En la situación menos

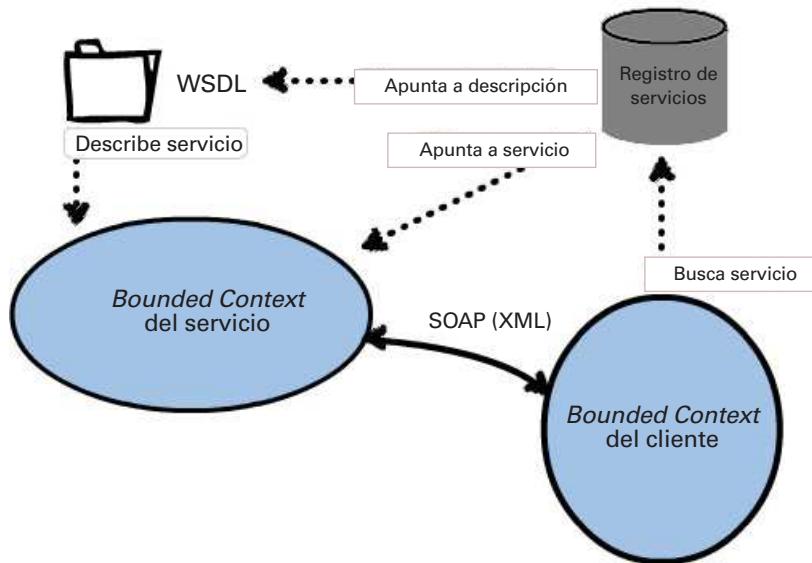
## Cómo hacer uso apropiado del *Context Mapping*

favorable, podría producirse la obligación de utilizar integración de bases de datos o mediante sistema de archivos, pero se espera que no sea así. La integración de bases de datos debe evitarse, pero si resulta imprescindible integrar de esa manera, hay que asegurarse de aislar el modelo de consumo mediante una capa anticorrupción.

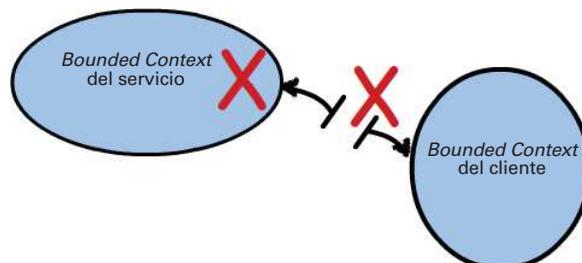


A continuación, una aproximación a tres de los tipos de integración más confiables. Primero, los enfoques de integración menos robustos, luego los más robustos; en ese orden, entonces, se comienza con RPC, seguido de RESTful HTTP y, finalmente, interfaz de mensajería.

## RPC con SOAP

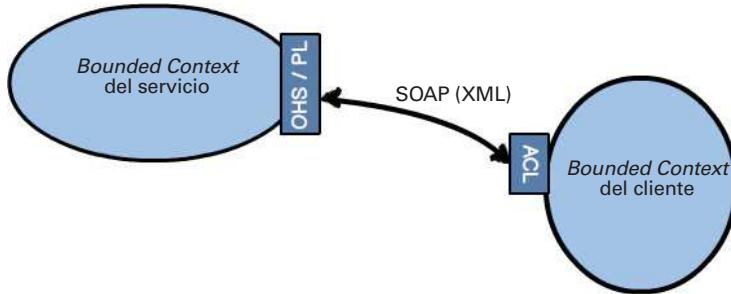


Las *llamadas a procedimientos remotos* (RPC, por su sigla original *Remote Procedure Calls*) pueden funcionar de varias maneras. Una forma popular mediante SOAP (*Simple Object Access Protocol*). La idea detrás de RPC con SOAP es que el uso de servicios de otro sistema parezca un procedimiento local sencillo o invocación de método. Aun así, la solicitud SOAP debe viajar a través de la red, llegar al sistema remoto, aplicarse con éxito y devolver resultados por la red. Esto conlleva el potencial de un fallo completo de red, o al menos, latencia no anticipada en el momento en que se implemente la integración por primera vez. Además, RPC sobre SOAP también implica un fuerte acoplamiento entre el *Bounded Context* del cliente y el *Bounded Context* que proporciona el servicio.



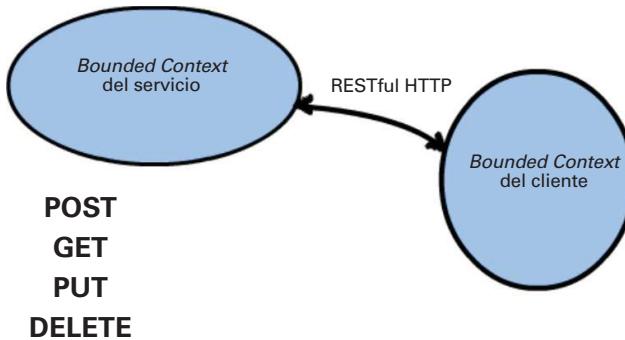
## Cómo hacer uso apropiado del *Context Mapping*

El principal problema con RPC, mediante SOAP u otro enfoque, es que puede carecer de solidez. Si hay un problema con la red o un problema con el sistema que aloja la API de SOAP, su llamada de procedimiento, aunque simple a priori, fallará por completo, dando sólo resultados de error. No dejarse engañar por la aparente facilidad de uso.

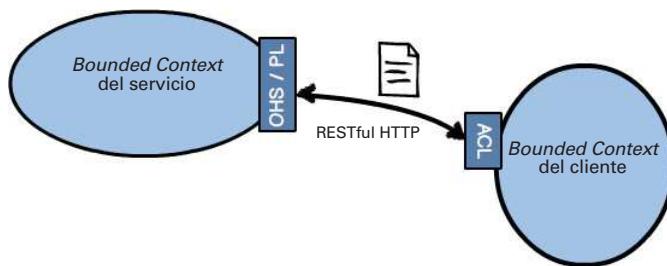


Cuando RPC funciona, y normalmente es así, puede ser una forma muy útil de integración. Si se pudiera influir en el diseño del *Bounded Context* del servicio, sería óptimo si existiera una API bien diseñada que proporcione un *servicio abierto* con un *idioma publicado (PL)*. De cualquier manera, el *Bounded Context* del cliente puede diseñarse con una *capa anticorrupción (ACL)* en el diagrama para aislar el modelo de influencias externas no deseadas.

## RESTful HTTP

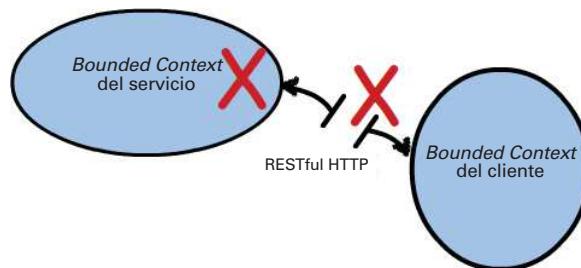


La integración con RESTful HTTP se concentra en los recursos que se intercambian entre *Bounded Contexts*, así como en las cuatro operaciones principales: POST, GET, PUT y DELETE. Muchos encuentran que el enfoque REST para la integración funciona bien porque ayuda a definir buenas API para computación distribuida. Es difícil argumentar contra esta afirmación dado el éxito de internet y de la web.

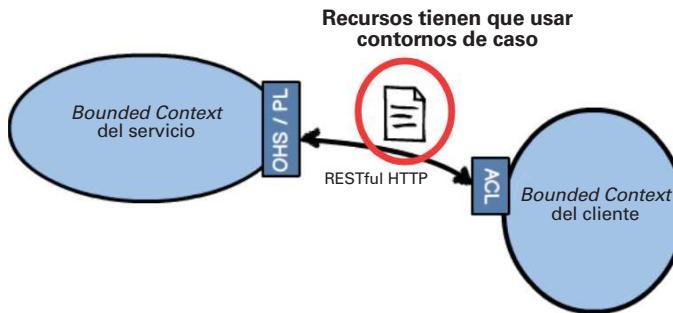


Existe una manera de trabajar bastante específica cuando se usa RESTful HTTP. En este libro no se darán detalles al respecto, pero hay que examinarlo antes de usar REST. El libro *REST in Practice* [RiP] es una buena obra para comenzar.

Un *Bounded Context* de servicio que tenga una interfaz REST debe proporcionar un *servicio abierto* (*OHS*) y un idioma publicado (*PL*). Los recursos merecen ser definidos como idioma publicado, y combinados con los REST URIs, formarán un servicio abierto natural.

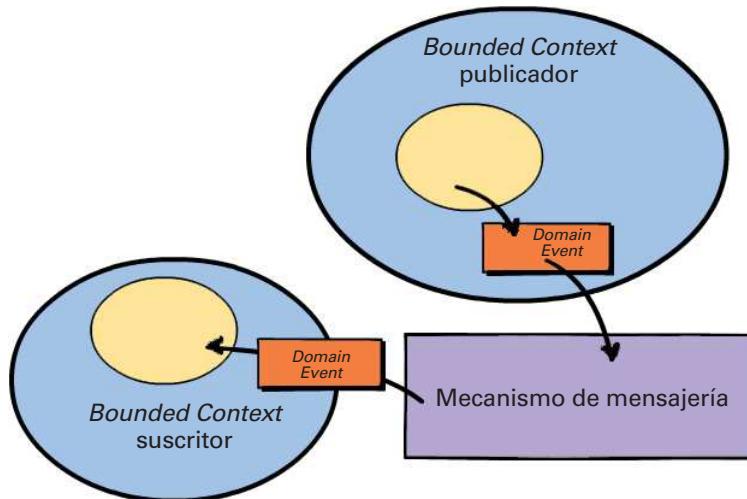


RESTful HTTP tiende a fallar por muchas de las mismas razones que RPC: fallos de proveedor de servicios y de la red, o latencia no prevista. Sin embargo, RESTful HTTP se basa en la premisa de internet, y ¿quién puede encontrar fallos en el historial de la web, en cuanto a confiabilidad, escalabilidad y éxito en general?



Un error común al usar REST es diseñar recursos que reflejan directamente los *agregados* en el modelo de dominio. Esto obliga a cada cliente a una relación *conformista*, donde, si ocurre cualquier cambio de forma en el modelo, los recursos se verán afectados de la misma manera. No es aconsejable. Óptimamente, los recursos deben diseñarse en forma sintética para seguir los casos de uso del cliente. Por “sintético” quiere decir que, para el cliente, los recursos proporcionados deben tener la forma y composición de lo que éste necesita, y no seguir la forma del modelo de dominio. A veces, el modelo se verá como lo que necesita el cliente, pero su necesidad es lo que impulsa el diseño de los recursos y no la composición actual del modelo.

## Mensajería



Cuando se utiliza mensajería asíncrona para integrar, se puede lograr mucho mediante el *Bounded Context* del cliente suscribiendo a los *Domain Events* publicados por sí mismo o por otro *Bounded Context*. El uso de mensajes es una de las formas más sólidas de integración porque elimina gran parte del acoplamiento temporal asociado al bloqueo de formularios RPC y REST, por ejemplo. Al anticipar la latencia del intercambio de mensajes se tiende a construir sistemas más robustos porque nunca se esperan resultados inmediatos.

---

### REST asíncrono

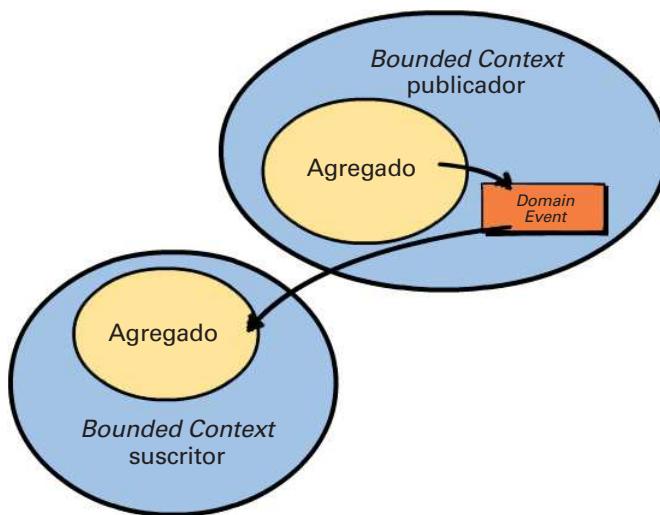
Es posible lograr mensajería asíncrona consultando de forma continua un conjunto de recursos REST que aumentan secuencialmente. Al usar procesamiento en segundo plano, un cliente sondearía continuamente un servicio que retornara recursos de tipo Atom Feed, como los de un blog, a modo de *Domain Event*. Éste es un enfoque seguro para mantener operaciones asíncronas entre un servicio y los clientes, y a la vez proporcionar eventos actualizados que continúan ocurriendo en el servicio. Si éste no está disponible por alguna razón, los clientes simplemente reintentarián en intervalos normales, o esperarían reintentar cuando el servicio estuviera nuevamente disponible. Este enfoque se analiza en detalle en *Implementing Domain-Driven Design* [IDDD].

---

### Evitar choques de trenes de integración

Cuando un *Bounded Context* del cliente (C1) se integra a un *Bounded Context* del servicio (S1), C1 generalmente no debería realizar una solicitud de bloqueo síncrono a S1 como resultado directo de la gestión de una solicitud hecha con ese propósito. Es decir, mientras que otro cliente (C0) realiza una solicitud de bloqueo a C1, no permitir que C1 realice una solicitud de bloqueo a S1. Permitirlo acarrea un potencial muy alto de causar un choque de trenes de integración entre C0, C1 y S1. Esto puede evitarse mediante el uso de mensajería asíncrona.

---



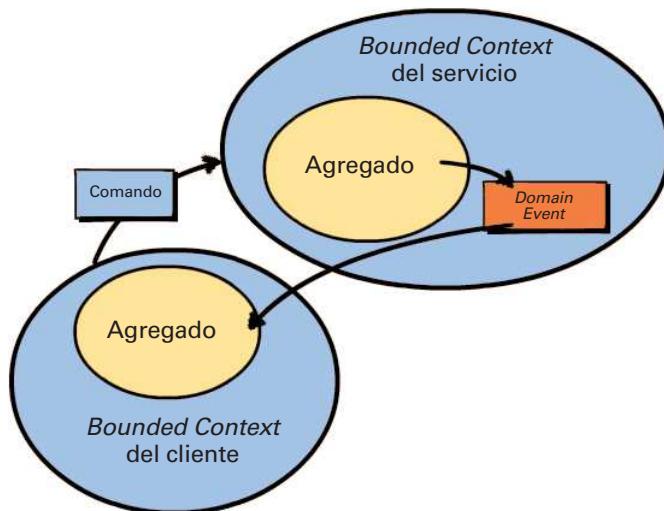
Normalmente, un *agregado* en un *Bounded Context* publica un *Domain Event* que podría ser consumido por cualquier parte interesada. Cuando un *Bounded Context* suscrito recibe ese *Domain Event*, se tomará una acción según su tipo y valor. Normalmente causará que se cree un nuevo *agregado* o que se modifique un *agregado* existente dentro del *Bounded Context*.

---

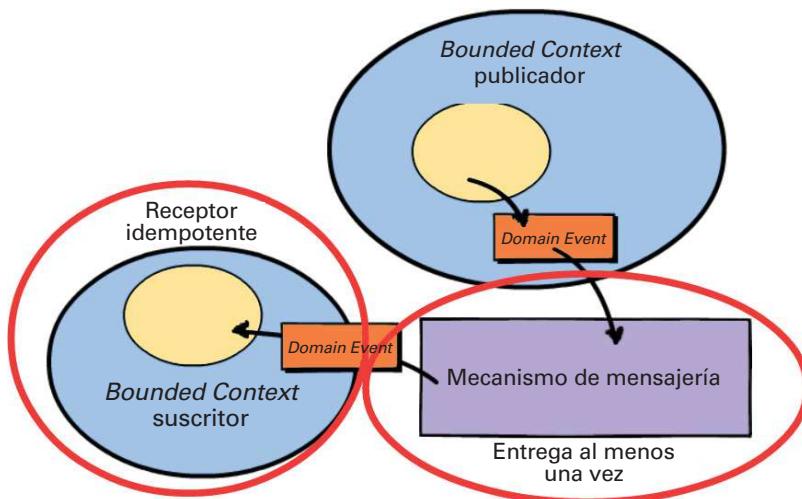
### ¿Son conformistas los consumidores de *Domain Event*?

Puede que surja la pregunta cómo los *Domain Events* pueden ser consumidos por otro *Bounded Context* sin forzar ese *Bounded Context* consumidor a una relación conformista. Como se recomienda en *Implementing Domain-Driven Design* [IDDD], específicamente en el capítulo 13, “Integración de *Bounded Contexts*”, los consumidores no deben usar los tipos de eventos (por ejemplo, clases) de un productor de eventos. Más bien, deben depender sólo del esquema de eventos, es decir, de su idioma publicado. Esto generalmente significa que si los eventos se publican como JSON, o quizás un formato de objeto más económico, el consumidor debe consumir los eventos analizándolos para obtener sus atributos de datos.

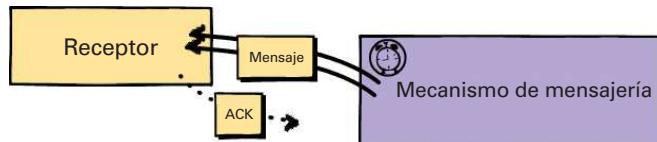
---



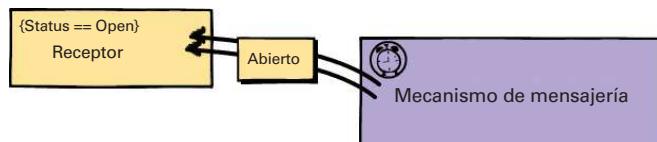
Por supuesto, lo anterior supone que un *Bounded Context* suscrito siempre puede beneficiarse de sucesos no solicitados en el *Bounded Context* de publicación. Sin embargo, a veces, un *Bounded Context* del cliente debe enviar proactivamente un mensaje de comando (*Command Message*) a un *Bounded Context* del servicio para forzar alguna acción. En tales casos, el *Bounded Context* del cliente recibirá un resultado como un *Domain Event* publicado.



En todos los casos de uso de mensajería para la integración, la calidad global de la solución dependerá en gran medida de la calidad del mecanismo de mensajería elegido. El mecanismo de mensajería debe admitir entrega al menos una vez (*At-Least-Once Delivery*) [Reactive] para garantizar que todos los mensajes se reciban. Esto también significa que el *Bounded Context* suscrito debe implementarse como un receptor idempotente (*Idempotent Receiver*) [Reactive].



La entrega al menos una vez [Reactive] es un patrón de mensajería donde el mecanismo de mensajería volverá a enviar periódicamente un mensaje determinado. Esto se hará en caso de pérdida de mensajes, receptores de reacción lenta o caídos, y cuando los receptores no reconocen recepción. Gracias a este diseño de mecanismo de mensajería, es posible entregar un mensaje más de una vez, aunque el remitente lo haya enviado sólo una vez. Sin embargo, esto no debe ser un problema si el receptor ya estuviera diseñado para tratar con este tipo de situaciones.



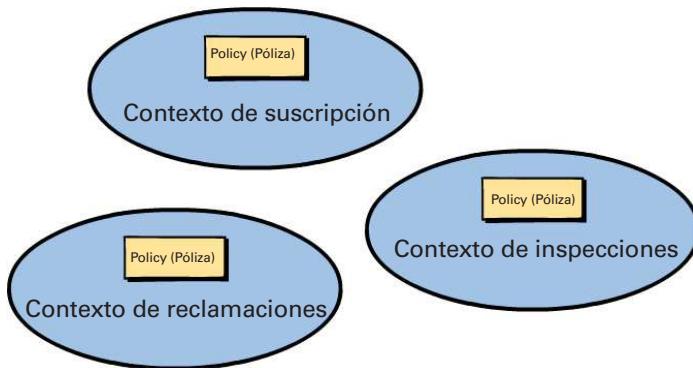
Siempre que un mensaje pueda ser entregado más de una vez, el receptor debe estar diseñado para tratar correctamente esta situación. El receptor idempotente [Reactive] describe cómo el receptor de una solicitud realiza una operación, de tal manera que produce el mismo resultado, incluso si se realizase varias veces. Por tanto, si un mismo mensaje se recibe varias veces, el receptor podrá tratarlo de manera segura. Esto puede significar que el receptor utiliza de-duplicación e ignora el mensaje repetido, o vuelve a aplicar la operación de forma segura con los mismos resultados que causó la entrega anterior.

Debido a que los mecanismos de mensajería siempre introducen comunicaciones asíncronas solicitud-respuesta (*Request-Response*) [Reactive], cierta latencia es común y esperada. Las solicitudes de servicio no

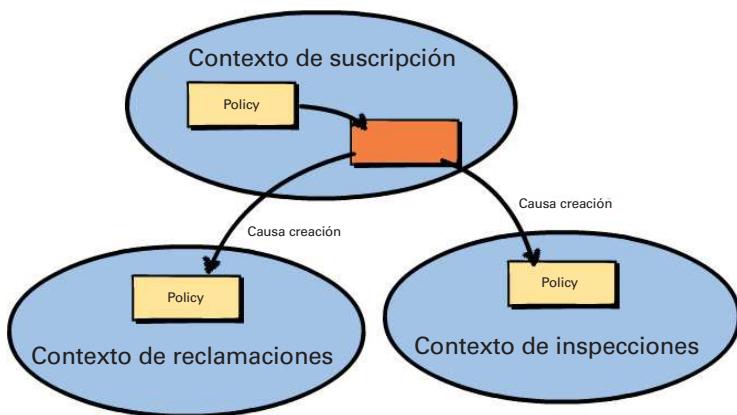
deben bloquearse (casi) nunca hasta que se cumpla el servicio. Por tanto, diseñar teniendo en cuenta la mensajería significa que siempre se planificará al menos algo de latencia, lo que hará que la solución general sea mucho más sólida desde el principio.

## Ejemplo de *Context Mapping*

---

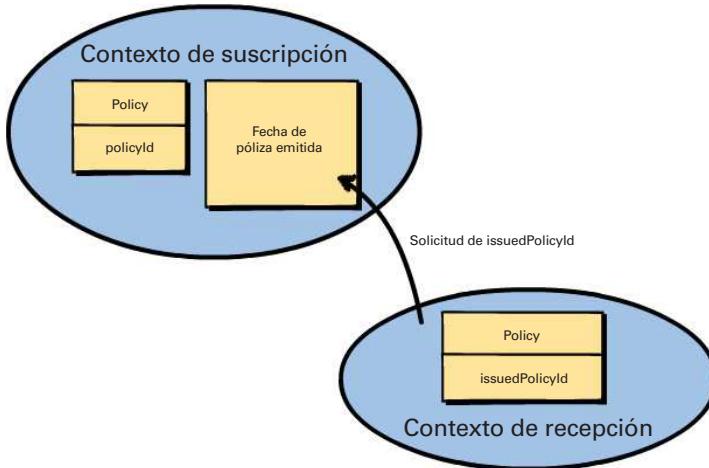


Retomando el ejemplo analizado en el capítulo 2, “Diseño estratégico con *Bounded Contexts* y el *Ubiquitous Language*”, surge una pregunta sobre la ubicación del tipo de *Policy* oficial. Recuérdese que existen tres tipos de *Policy* diferentes en tres *Bounded Contexts* distintos. Entonces, ¿dónde vive la “póliza principal” en la empresa de seguros? Es posible que pertenezca a la división de suscripciones, ya que es donde se origina. Para seguir con el ejemplo, dígase que pertenece a la división de suscripciones. Entonces, ¿cómo aprenden los otros *Bounded Contexts* sobre su existencia?



Cuando se emite un componente de tipo `Policy` en el contexto de suscripciones (*Underwriting Context*), podría publicar un *Domain Event* llamado `PolicyIssued`.

Mediante una suscripción por mensajería, cualquier otro *Bounded Context* puede reaccionar a ese *Domain Event*, que podría causar la creación de un componente de `Policy` correspondiente en el *Bounded Context* suscrito.



El *Domain Event PolicyIssued* contendría la identidad de la *Policy* oficial, en este caso *policyId*. Cualquier componente creado en un *Bounded Context* escuchando esos eventos mantendría esa identidad para su trazabilidad hasta el *contexto de suscripciones original*. En este ejemplo, la identidad se guarda como *issuedPolicyId*. Si se necesitan más datos de la *Policy* de los que proporciona el *Domain Events PolicyIssued*, el *Bounded Context* que recibe el evento siempre puede volver a consultar el *contexto de suscripciones* para obtener más información. En este caso, el *Bounded Context* que escucha utiliza el *issuedPolicyId* para realizar una consulta al contexto de suscripciones.

---

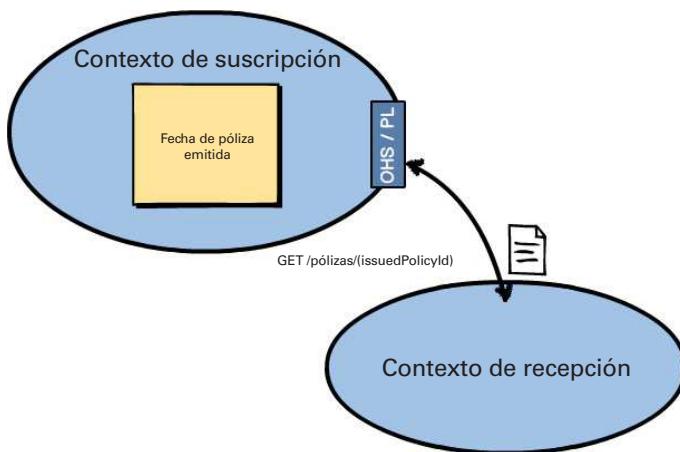
### Pros y contras de enriquecer o consultar el contexto original

A veces, es ventajoso enriquecer los *Domain Events* con datos suficientes para satisfacer las necesidades de todos los consumidores. Otras veces, es mejor tener *Domain Events* más reducidos, y permitir que se puedan consultar cuando los consumidores necesitan más datos. En la primera opción (enriquecer), se permite mayor autonomía de consumidores dependientes. Si la autonomía es un requisito principal, se puede considerar esta estrategia.

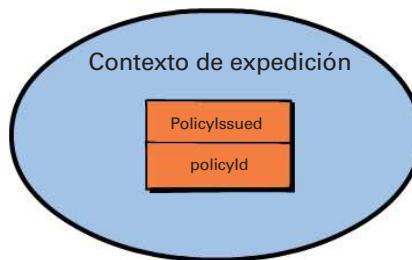
Por otro lado, es difícil predecir todos los datos que necesitarán los consumidores en los *Domain Events*, y puede haber demasiado enriquecimiento si lo proporciona todo. Por ejemplo, enriquecer en gran medida los *Domain Events* puede ser una mala elección en términos de seguridad. Si es el caso, un diseño de *Domain Events* ligero y un modelo de consulta completo y seguro, disponible para solicitudes de consumidores, puede ser la opción necesaria.

Ciertas circunstancias requieren combinación equilibrada de ambos enfoques.

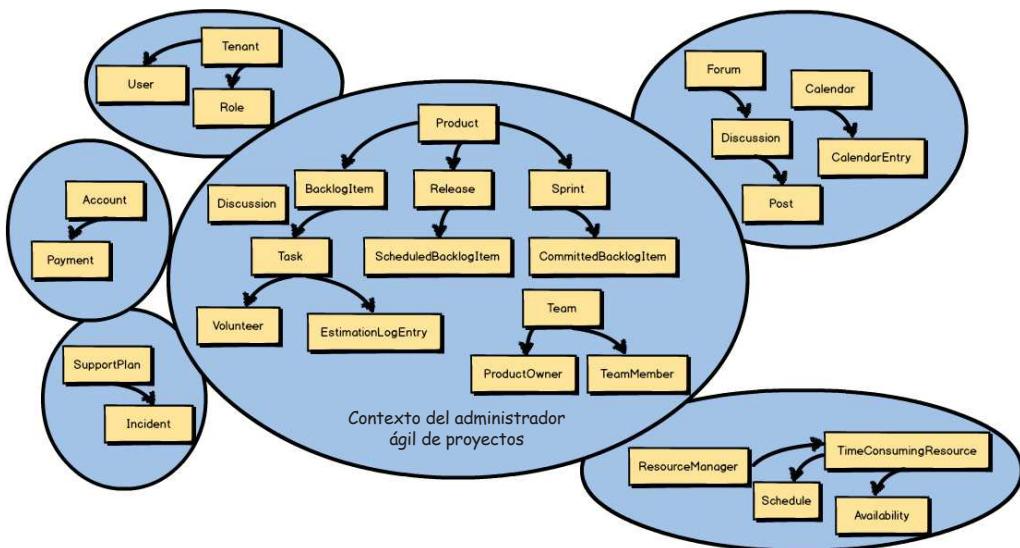
---



¿Cómo podría funcionar la estrategia de consultar al contexto original (*Query-Back*) considerando el contexto de las expediciones? Es posible diseñar un servicio abierto (*Open Host Service*) RESTful y un idioma publicado en el contexto de suscripciones. Un sencillo HTTP GET con `issuedPolicyId` extraería el `IssuedPolicyData`.



Probablemente surja la pregunta acerca de los datos del *Domain Event* del ejemplo anterior `PolicyIssued`. En el capítulo 6, “Diseño táctico con *Domain Event*” se abordarán los detalles del diseño de *Domain Event*.



¿Qué sucedió con el ejemplo sobre el contexto del administrador ágil de proyectos? Alternar entre éste y el dominio del negocio de seguros permite examinar DDD mediante múltiples ejemplos. Esto debe ayudar a comprender aún mejor DDD. En el siguiente capítulo se abordará nuevamente dicho contexto.

## Resumen

---

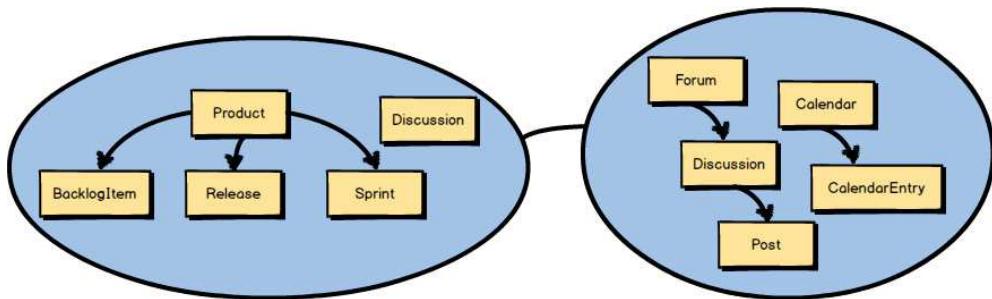
En síntesis, lo aprendido:

- Los diversos tipos de relaciones de *Context Mapping*, como alianza, cliente-proveedor y capa anticorrupción.
- Cómo utilizar la integración de *Context Mapping* con RPC, con RESTful HTTP y con mensajería.
- Cómo funcionan los *Domain Events* con mensajería.
- Una base mediante la cual se puede consolidar la experiencia de *Context Mapping*.

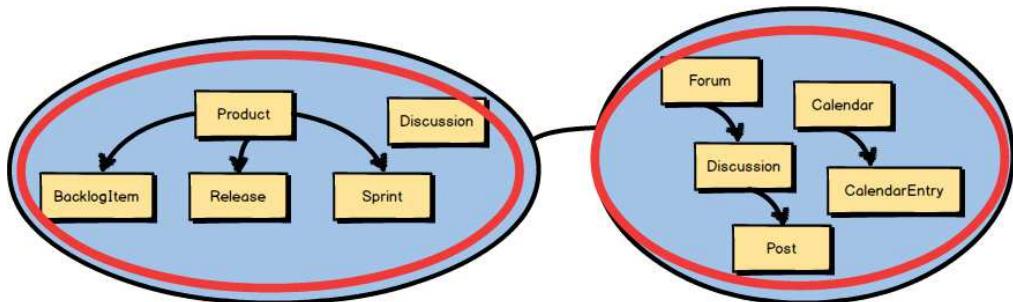
Para una mayor cobertura de *Context Maps*, véase el capítulo 3 sobre *Implementing Domain-Driven Design* [IDDD].

# CAPÍTULO 5

## Diseño táctico con agregados

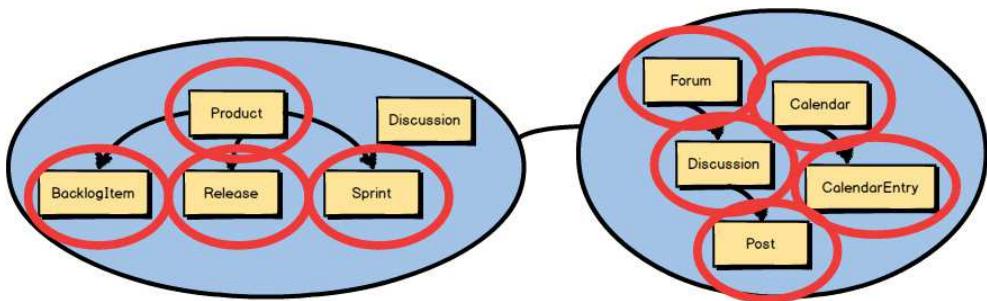


Hasta ahora se ha analizado el diseño estratégico con *Bounded Contexts*, subdominios y *Context Maps*. Aquí se pueden observar dos *Bounded Contexts*: el dominio principal denominado Contexto de *Agile Project Management* y un dominio de soporte que proporciona herramientas de colaboración mediante integración con *Context Mapping*.



Pero, ¿qué pasa con los conceptos que viven dentro de un *Bounded Context*? Estos temas ya se han tratado, aunque a continuación se cubrirán con más detalle. Es probable que sean los agregados del modelo por desarrollar.

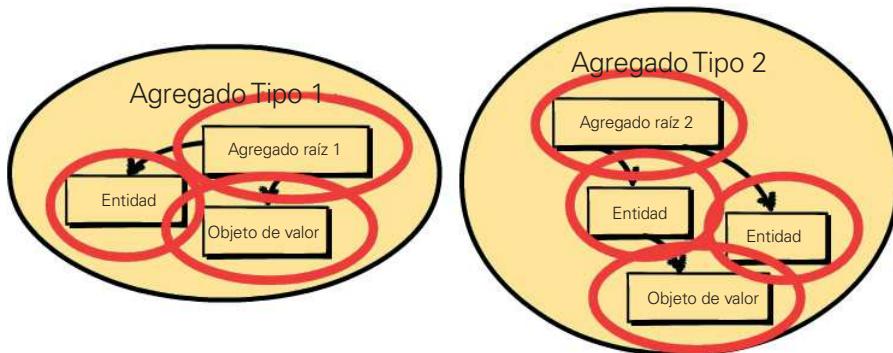
## ¿Por qué se utilizan?



Cada uno de los conceptos señalados en rojo dentro de estos dos *Bound-ed Contexts* son agregados. El único concepto que no está resaltado—*Discussion*—se modela como un *objeto de valor*. Aun así, el enfoque se hará en los *agregados* en este capítulo, y se analizará con mayor cuidado cómo modelar *Product*, *BacklogItem*, *Release* y *Sprint*.

### ¿Qué es una entidad (*Entity*)?

Una *entidad* modela una cosa individual y cada una tiene una identidad única que puede distinguir su individualidad de todas las demás *entidades* del mismo o diferente tipo. Muchas veces, incluso la mayoría de ocasiones, una *entidad* será mutable; es decir, su estado cambiará con el tiempo. Sin embargo, aunque una *entidad* no es necesariamente mutable, puede ser inmutable. La característica principal de una *entidad* frente a otras herramientas de modelado es su singularidad, su individualidad. Se sugiere consultar *Implementing Domain-Driven Design* [IDDD], donde se cubre todo lo relacionado con las *entidades* en forma más detallada.

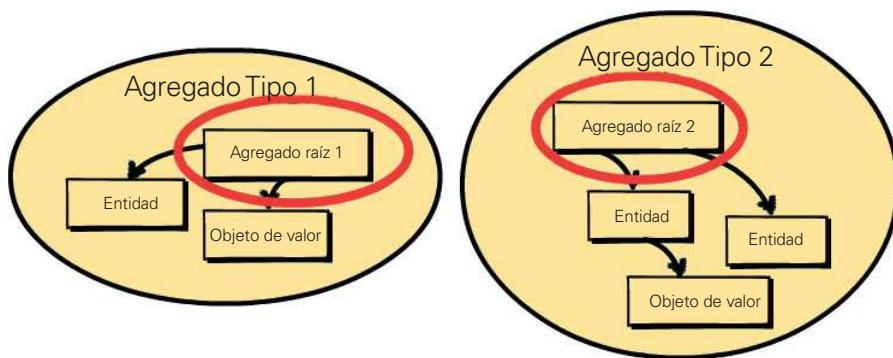


¿Qué es un *agregado*? Dos aparecen representados aquí. Cada *agregado* se compone de una o más entidades, y una entidad se conoce como *agregado raíz*. Los *agregados* también pueden tener *objetos de valor* compuestos en ellos. Tal como se observa, los *objetos de valor* se utilizan dentro de ambos *agregados*.

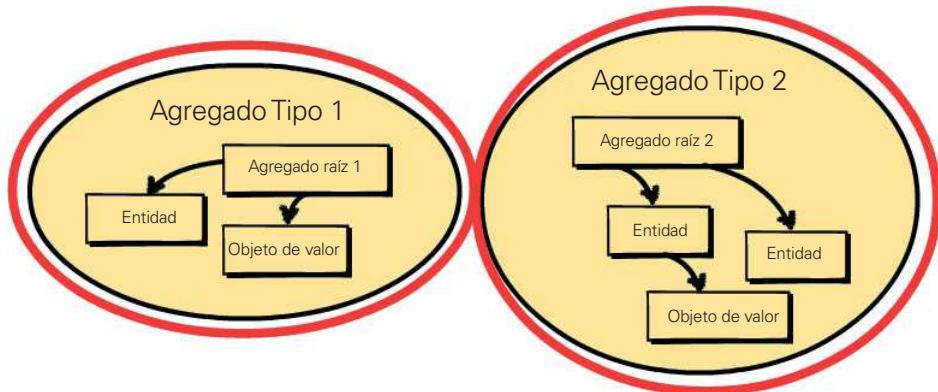
### ¿Qué es un objeto de valor (*Value Object*)?

Un objeto de valor, o simplemente un valor, modela un conjunto conceptual inmutable. Dentro del modelo, *el valor* es solo eso, un valor. A diferencia de una entidad, no tiene una identidad única, y la equivalencia se determina comparando los atributos encapsulados por el tipo de valor. Además, un objeto de valor no es una cosa, sino que a menudo se usa para describir, cuantificar o medir una entidad.

Se sugiere consultar *Implementing Domain-Driven Design* [Iddd], donde se cubre todo lo relacionado con los objetos de valor en forma más detallada.



La *entidad raíz* de cada *agregado* posee todos los demás elementos agrupados en su interior. El nombre de la entidad raíz es el nombre conceptual del *agregado*. Debe elegir un nombre que describa correctamente el conjunto conceptual que *modela* el agregado.



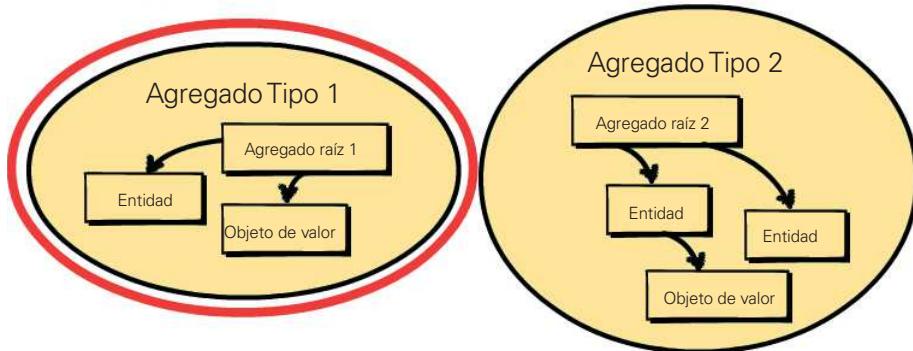
Cada *agregado* forma un límite de consistencia transaccional. Esto significa que dentro de un único *agregado*, todas las partes compuestas deben ser coherentes, de acuerdo con las reglas de negocio, cuando la transacción de control se confirma en la base de datos. Esto no significa que no se deben componer otros elementos dentro de un *agregado* que no necesitan ser consistentes después de una transacción. Después de todo, un agregado también modela un todo conceptual. Principalmente, debes estar enfocado en la consistencia transaccional. El límite exterior trazado alrededor de Agregado Tipo 1 y Agregado Tipo 2 representa una transacción separada que tendrá el control de persistir atómicamente cada clúster de objetos.

### Significado amplio de transacción

Hasta cierto punto, el uso de transacciones en la aplicación constituye un detalle de implementación. Por ejemplo, un uso típico tendría un servicio de aplicación [IDDD] controlando la transacción atómica de la base de datos en representación del modelo de dominio. Bajo una arquitectura diferente, e.g. el **Modelo actor (Actor model)** [reactivo], donde cada *agregado* se implementa como un actor, las transacciones podrían manejarse utilizando **Event Sourcing** (véase siguiente capítulo) con una base de datos que no admitiera transacciones atómicas. En todo caso, lo que se quiere decir con “transacción” es cómo se aíslan las modificaciones a un *agregado*, y cómo las invariantes de negocio (las reglas a las que siempre debe adherirse el software) tienen consistencia garantizada después de cada operación de negocio. Ya sea este requisito controlado por una transacción atómica de base de datos o por algún otro medio, el estado del *agregado*, o su representación por medio de **Event Sourcing**, debe transicionarse y mantenerse de manera segura y correcta en todo momento.

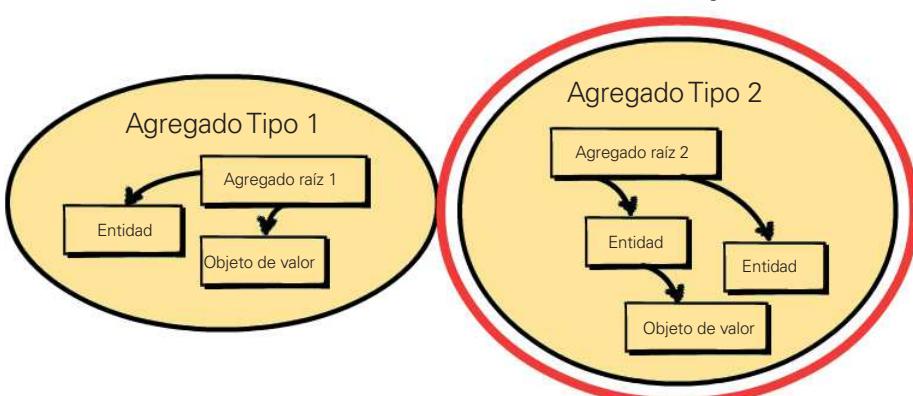
Las razones para el límite transaccional están motivadas por el negocio, ya que éste es el que determina qué debe ser un estado válido del clúster en determinado momento. En otras palabras, si el *agregado* no se almacena en un estado completo y válido, la operación de negocio que se realiza se consideraría incorrecta de acuerdo con las reglas del negocio.

### Transacción sencilla

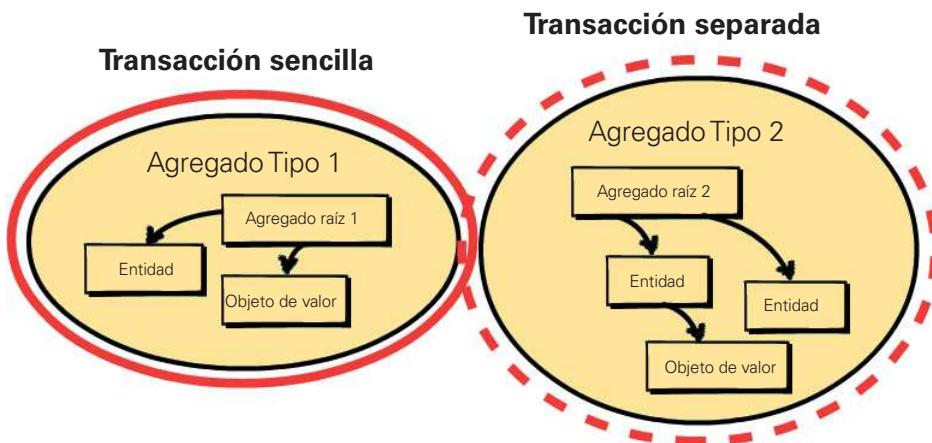


Para pensar en esto de una manera diferente, considera lo siguiente: aunque aquí se representan dos *agregados*, solo uno de ellos debe comprometerse en una sola transacción. Esa es regla general del diseño de *agregados*: modifica y confirma solo una instancia de *agregado* en cada transacción; por ello, solo se observa la instancia de Agregado Tipo 1 dentro de una transacción. Las otras reglas del diseño de *agregados* se abordarán pronto.

### Transacción separada



Cualquier otro *agregado* se modificará y se confirmará en una transacción específica. Por eso se dice que un *agregado* es un límite de consistencia transaccional. Así, diseñar las composiciones *agregadas* de tal forma que permitan la coherencia y éxito transaccional. Tal como se observa aquí, una instancia Agregado Tipo 2 se controla mediante una transacción separada de la instancia Agregado Tipo 1.



Dado que las instancias de estos dos *agregados* están diseñadas para ser modificadas en transacciones separadas, ¿cómo conseguir que la instancia Agregado Tipo 2 se actualice en función de los cambios realizados en la instancia Agregado Tipo 1, a los que debe reaccionar el modelo de dominio propuesto? Pregunta apropiada para abordar y responder poco más adelante en este mismo capítulo. El punto principal por recordar de esta sección es que las reglas de negocio son los drivers para determinar qué debe estar completo y coherente al final de una transacción.

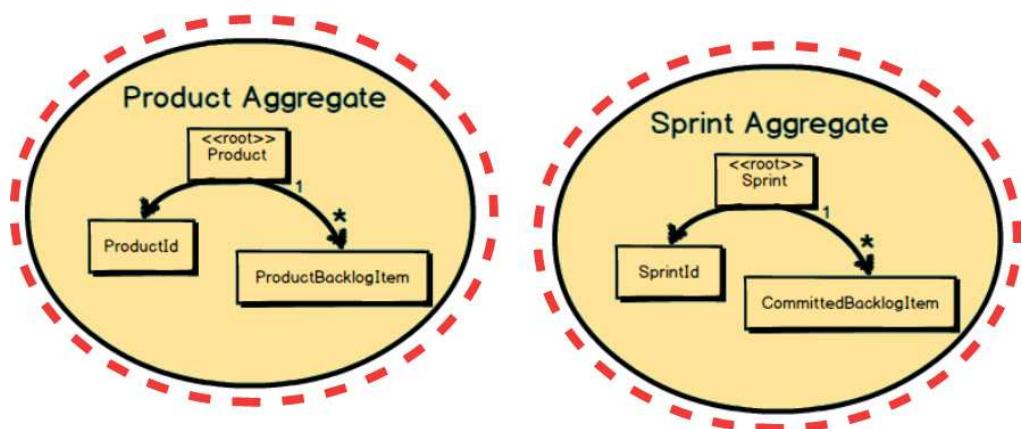
## Reglas básicas de los agregados

Obsérvense a continuación las cuatro reglas básicas del diseño de agregados :

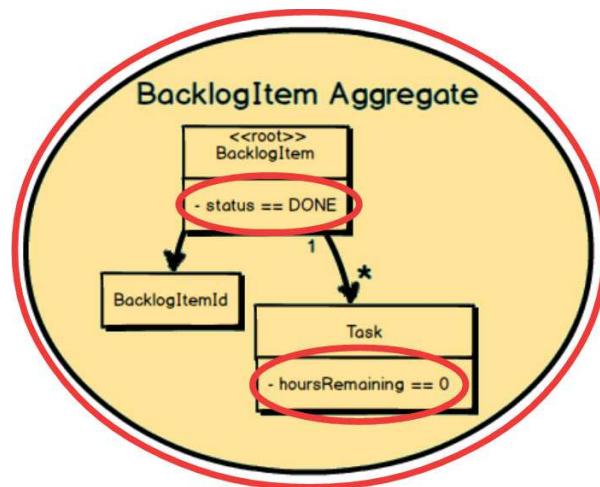
1. Proteger las invariantes del negocio dentro de límites del agregado.
2. Diseñar *agregados* simples.
3. Referenciar otros *agregados* usando únicamente su identidad.
4. Actualizar otros *agregados* usando *Eventual Consistency*.

Por supuesto, estas reglas no son controladas estrictamente por ningún “vigilante de DDD”. Sirven como buena guía para que, cuando sean aplicadas cuidadosamente, contribuyan a diseñar *agregados* que funcionen de manera efectiva. A continuación, entonces, se profundizará en cada una de estas reglas para observar cómo deben aplicarse siempre que sea posible.

## Regla 1: proteger las invariantes del negocio dentro de límites del agregado

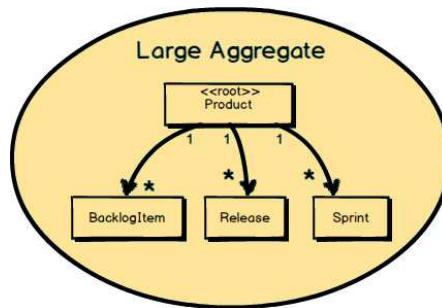


La regla 1 significa que la empresa debe ser, en última instancia, quien determine las composiciones de *agregados* según lo que debe ser consistente cuando se confirma una transacción. En el ejemplo de la página 81, Product está diseñado de tal manera que al final de una transacción, todas las instancias de ProductBacklogItem deben contabilizarse y ser coherentes con la *raíz* de Product. Además, Sprint está diseñado de tal manera que al final de una transacción, todas las instancias compuestas de CommittedBacklogItem deben contabilizarse y ser coherentes con la *raíz* de Sprint.



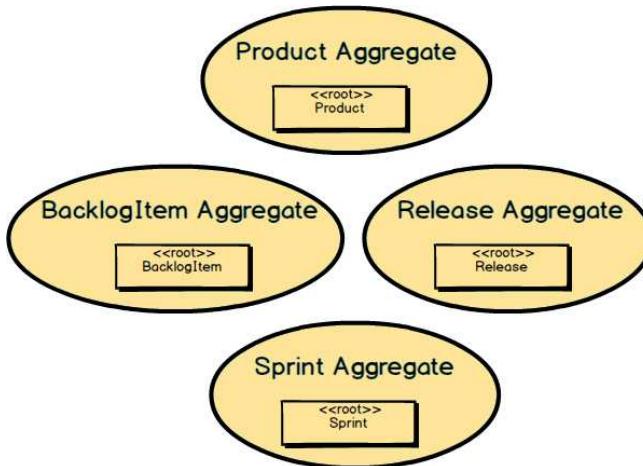
Para aclarar un poco más la regla 1, otro ejemplo. Obsérvese aquí el **Backlog-Item Aggregate**. Hay una regla de negocios que dice: “Cuando todas las instancias de *Task*, tienen `hoursRemaining`, igual a cero, el estado de `BacklogItem` se debe establecer en `DONE`. Por tanto, al final de una transacción se debe cumplir esta invariante específica del negocio, pues este lo requiere.

## Regla 2: diseñar agregados simples



Esta regla resalta que la huella de memoria y el alcance transaccional de cada *agregado* deben ser relativamente pequeños. En el diagrama anterior,

el *agregado* que se representa no es pequeño. El *Product* contiene una colección potencialmente muy grande de instancias de *BacklogItem*, una gran colección de instancias de *Release* y una gran colección de instancias *Sprint*. Con el tiempo, estas colecciones podrían llegar a ser bastante grandes, con miles de instancias de *BacklogItem* y probablemente cientos de instancias de *Release* y *Sprint*. Este enfoque de diseño, por lo general, no es una buena opción.



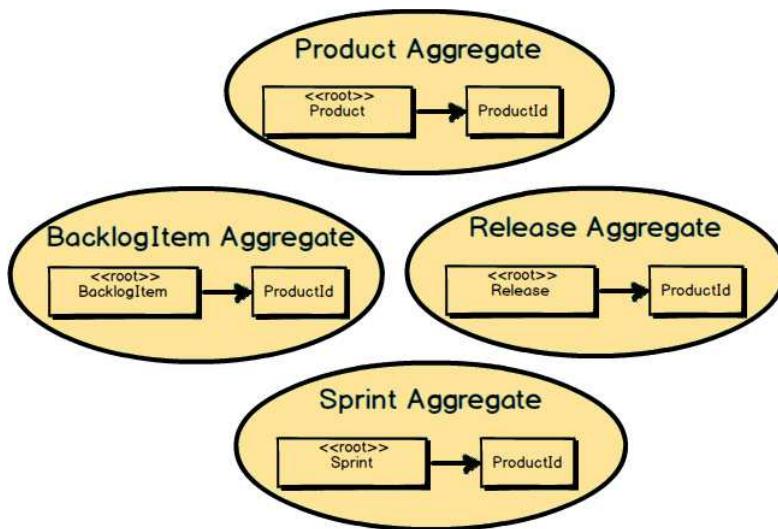
Sin embargo, si se divide el *agregado Product* para formar cuatro agregados separados, se obtiene lo siguiente: un *agregado Product* simple, un *agregado Backlogitem* simple, un *agregado Release* simple y un *agregado Sprint* simple. Estos se cargan rápidamente, requieren menos memoria y son más rápidos en recolección de basura (*garbage collect*). Quizás lo más importante es que estos *agregados* tendrán éxito transaccional con mucha más frecuencia que el anterior gran clúster *agregado Product*.

Seguir esta regla tiene el beneficio adicional que hará cada *agregado* más fácil de trabajar, ya que cada tarea asociada podrá ser administrada por un solo desarrollador. También significa que el *agregado* será más fácil de pasar a pruebas (testear).

Otro punto para tener en cuenta al diseñar agregados es el principio de responsabilidad única (*Single Responsibility Principle*, SRP). Si el *agregado* trata de hacer demasiadas cosas, no sigue el SRP, y es probable que esto se refleje en su tamaño. Cuestionarse, por ejemplo, si el *Product* es una

implementación muy enfocada de un producto Scrum, o si también intenta ser otra cosa. ¿Cuál es la razón para modificar el Product: convertirlo en un mejor producto Scrum, o para administrar, backlogitems, releases y sprints? Entonces, se debe modificar el producto solo para que sea un mejor producto Scrum.

### Regla 3: referenciar otros agregados usando únicamente su identidad



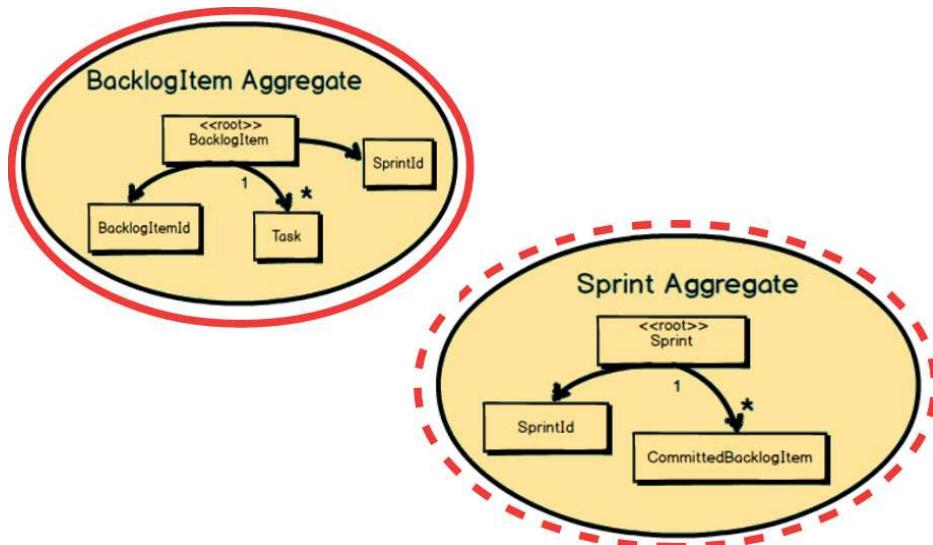
Ahora que se ha dividido el gran clúster Product en cuatro *agregados* más pequeños, ¿de qué manera cada uno de ellos debe hacer referencia a los otros cuando sea necesario? Aquí se sigue la regla 3, «Referenciar otros agregados usando únicamente su identidad». En este ejemplo, se observa que tanto Backlogitem, Release y Sprint hacen referencia a Product al tener un IdProduct. Esto ayuda a mantener los *agregados* simples y evita tener que modificar varios *agregados* en la misma transacción.

Asimismo, esto contribuye a mantener el diseño de los *agregados* simple y eficiente, reduce los requerimientos de memoria y acelera la carga desde un almacenamiento de persistencia. Además, ayuda a cumplir la regla de no modificar otras instancias de *agregados* en la misma transac-

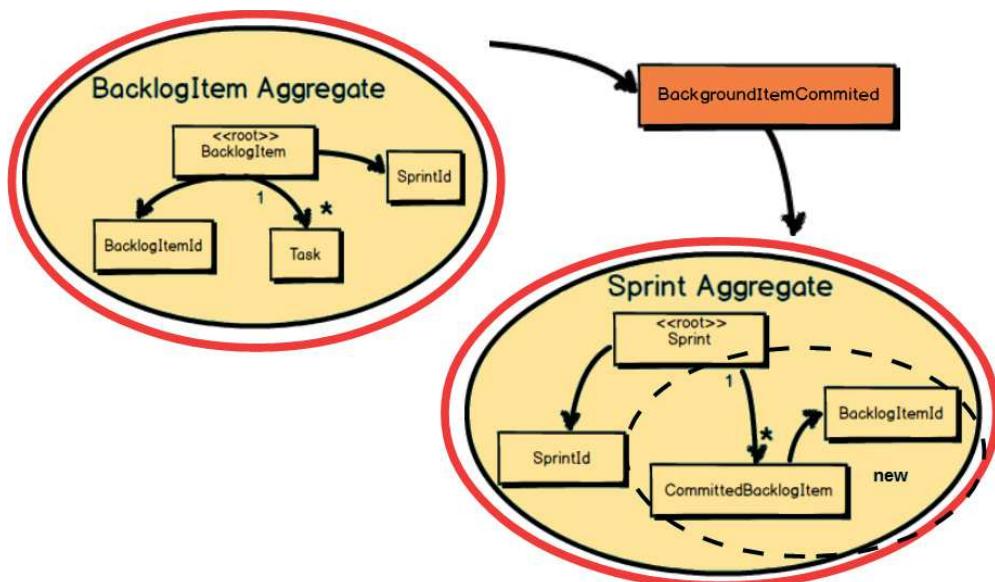
ción. Con solo las identidades de otros agregados, no hay manera fácil de obtener una referencia directa de objeto para estos.

Otro beneficio de usar referencia por identidad es que los *agregados* se pueden almacenar fácilmente en casi cualquier tipo de mecanismo de persistencia, como base de datos relacional, base de datos de documentos, almacenamiento de valores clave (*key-values*) y rejillas/tejidos (*grids/fabrics*) de datos. Así, se abren varias opciones: usar una tabla relacional MySQL o almacenamiento basado en JSON, como PostgreSQL o MongoDB, GemFire/Geode, Coherence o GigaSpaces.

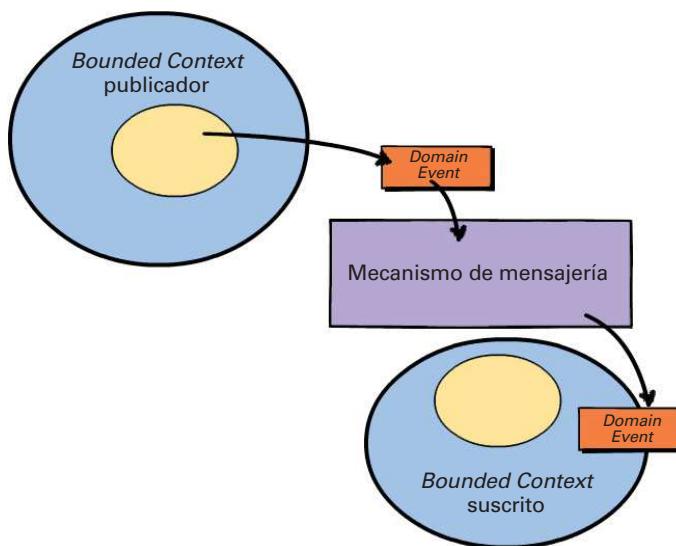
#### Regla 4: actualizar otros agregados usando *Eventual Consistency*



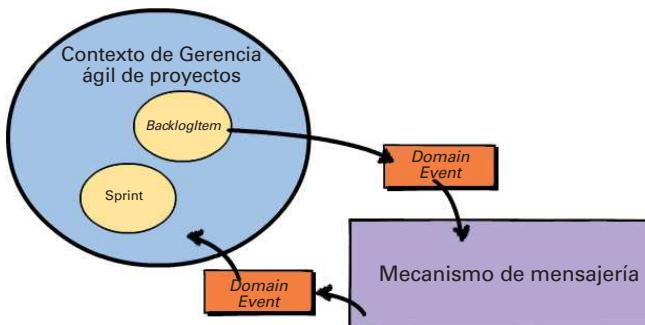
Aquí un BacklogItem está comprometido a un Sprint. Tanto el BacklogItem como el Sprint deben reaccionar a esto. El BacklogItem es el que sabe primero que se ha comprometido a un Sprint. Esto se gestiona en una transacción, cuando el estado del BacklogItem se modifica para contener el SprintId del Sprint con el que se ha comprometido. Entonces, ¿cómo asegurar que el Sprint también se actualice con el BacklogItemId del recién comprometido BacklogItem?



Como parte de la transacción agregada del BacklogItem, este publica un *Domain Event* llamado BacklogItemCommitted. La transacción BacklogItem se completa y su estado persiste junto con el *Domain Event* BacklogItemCommitted. Cuando el BacklogItemCommitted se dirige a un suscriptor local, se inicia una transacción y el estado de Sprint se modifica para mantener el BacklogItemId del committed BacklogItem. El Sprint mantiene el BacklogItemId en una nueva entidad BacklogItemCommitted.



Recuérdese ahora lo aprendido en el capítulo 4, “Diseño estratégico con *Context Mapping*”. Los *Domain Events* son publicados por un *agregado* y se suscriben a ellos *Bounded Contexts* interesados. El mecanismo de mensajería entrega los *Domain Events* a las partes interesadas mediante suscripciones. El *Bounded Context* interesado puede ser el mismo desde el que se publicó el *Domain Event*, o puede ser uno diferente.



En el caso del *BacklogItem* y el *agregado Sprint*, el editor y el suscriptor están en el mismo *Bounded Context*. No sería absolutamente necesario usar un producto de middleware de mensajería para este caso, pero es fácil hacerlo si ya se está usando para publicar en otros *Bounded Contexts*.

### Por si causa preocupación la *Eventual Consistency*

No existe nada difícil en el uso de la *Eventual Consistency*. No obstante, hasta que haya algo de experiencia, puede haber preocupación cuando se utiliza. Si es así, se debe particionar el modelo en *agregados* de acuerdo con los límites transaccionales definidos por el negocio. Sin embargo, no hay nada que impida realizar modificaciones a dos o más *agregados* en una sola transacción de base de datos atómica. Se puede adoptar este enfoque en casos en los que se sabe que tendrán éxito, y utilizar consistencia eventual para todos los demás. Esto permitirá acostumbrarse a técnicas sin dar un paso inicial muy grande. Simplemente, comprender que esta no es la forma habitual como se deben usar los *agregados* y, como resultado, experimentar fallas transaccionales.

## *Modelamiento de agregados*



Hay algunos “anzuelos” que esperan mientras se trabaja en un modelo de dominio, mediante la implementación de agregados. Un anzuelo grande y desagradable es el *Anemic Domain Model* (modelo de dominio anémico) [IDDD]. Sucede cuando se utiliza un modelo de dominio orientado a objetos, y todos los *agregados* tienen solo asesores públicos (*getters* y *setters*) sin un comportamiento real del negocio. Esto tiende a suceder cuando predomina el enfoque técnico sobre el enfoque empresarial durante la modelamiento. El diseño de un *Anemic Domain Model* requiere asumir todos los gastos generales y recargos que implica un modelo de dominio sin tener ninguno de sus beneficios. ¡No hay que caer en esa trampa!

Además, tener cuidado con la pérdida de lógica del negocio en los ser-

vicios de aplicación por encima del modelo de dominio. Puede ocurrir sin ser detectada, al igual que la atrofia física. Delegar la lógica de negocio de los servicios a las clases ayuda (*helper*)/*utility* tampoco va a funcionar bien. Las *utilities* de servicio siempre muestran una crisis de identidad y nunca pueden mantener su historial en orden. Hay que colocar la lógica del negocio en el modelo de dominio, o sufrir *bugs* patrocinados por un *Anemic Domain Model*.

## ¿Y la programación funcional?

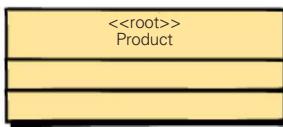
Al usar programación funcional, las reglas cambian en gran medida.

Si bien un *Anemic Domain Model* no es buena idea cuando se usa programación orientada a objetos, es muy frecuente cuando se aplica programación funcional. Esto porque la programación funcional promueve la separación de datos y el comportamiento. Los datos se diseñan como estructuras de datos inmutables o tipos de registros, y el comportamiento se implementa como funciones puras que operan en registros inmutables de tipos específicos. En lugar de modificar los datos que reciben las funciones como argumentos, las funciones devuelven valores nuevos, y estos últimos pueden constituir el nuevo estado de un *agregado* o un *Domain Event* que representa una transición en el estado de un *agregado*.

En este capítulo, hasta el momento en gran medida se ha abordado el enfoque orientado a objetos porque sigue siendo el más utilizado y comprendido. Sin embargo, si se emplea lenguaje funcional y un enfoque de DDD, tener en cuenta que parte de esta guía no es aplicable o, al menos, está sujeta a reglas superiores.

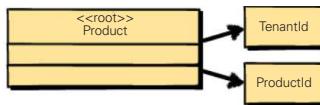


A continuación, se mostrarán algunos de los componentes técnicos necesarios para implementar un diseño de *agregado* básico. Se supone entonces que se utiliza Scala, C#, Java u otro lenguaje de programación orientado a objetos. Los siguientes ejemplos están en C#, aunque son comprensibles para Scala, F#, Java, Ruby, Python y otros programadores.



```
public class Product : Entity
{
    ...
}
```

Lo primero que se debe hacer es crear una clase para la *entidad agregada raíz*. Aquí se puede observar una representación UML (*Unified Modeling Language*, o *Lenguaje Unificado de Modelamiento*) de la *entidad Product raíz*. También se incluye la clase de Product en C#, que extiende una clase base denominada Entity. Esta clase base solo se ocupa de asuntos de entidad estándar. Véase *See Implementing Domain-Driven Design* [IDDD] para análisis exhaustivos sobre el diseño y la implementación de *entidades y agregados*.

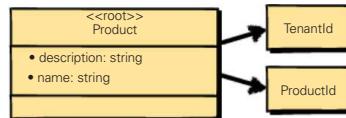


```
public class Product : Entity
{
    private ProductId productId;
    private TenantId tenantId;
}
```

Cada entidad agregada raíz debe tener una identidad global única. Un Product en el contexto de *núcleo ágil* de gerencia de proyectos (Agile Project Management), tiene en realidad dos formas de identidad global única. TenantId examina la *entidad raíz* en una organización de suscriptores determinada. (Toda organización que se suscribe a servicios ofrecidos se conoce como *inquilino* y, por consiguiente, por ello tiene una identidad única). La segunda identidad, que también es globalmente única, es ProductId. Esta segunda identidad distingue el Product de todos los demás, en la misma parcela (*tenant*). También se incluye código C# que declara las dos identidades dentro del Product.

## Uso de objetos de valor

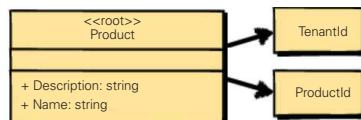
En esta instancia, tanto TenantId como ProductId se modelan como objetos de valor inmutables.



```

public class Product : Entity
{
    private string description;
    private string name;
    private ProductId productId;
    private TenantId tenantId;
}
  
```

A continuación, captura un atributo intrínseco o campos que sean necesarios para hallar el *agregado*. En el caso de Product, hay tanto description como name. Los usuarios pueden buscar uno o ambos para hallar cada Product. También se proporciona el código C # que declara estos dos atributos intrínsecos.



```

public class Product : Entity
{
    ...
    public string Description
    { get; private set; }

    public string Name
    { get; private set; }
}
  
```

Por supuesto, se pueden agregar comportamientos simples como *read accessors (getters)* para atributos intrínsecos. En C# esto se haría usando *getters* de propiedad pública. Sin embargo, es posible no exponer *setters*

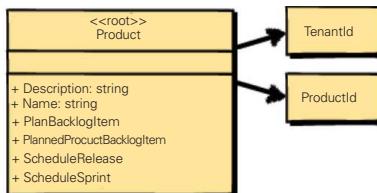
como públicos, y sin estos, ¿cómo cambian los valores de propiedad/atributo?

Cuando se utiliza un enfoque *orientado a objetos* (C#, Scala y Java), se cambia el estado interno utilizando métodos de comportamiento. Si se utiliza un enfoque funcional (F#, Scala y Clojure), las funciones devolverán nuevos valores que son diferentes de los valores pasados como argumentos.



```
public class Product : Entity
{
    ...
    public string Name
        { get; private set; }
}
```

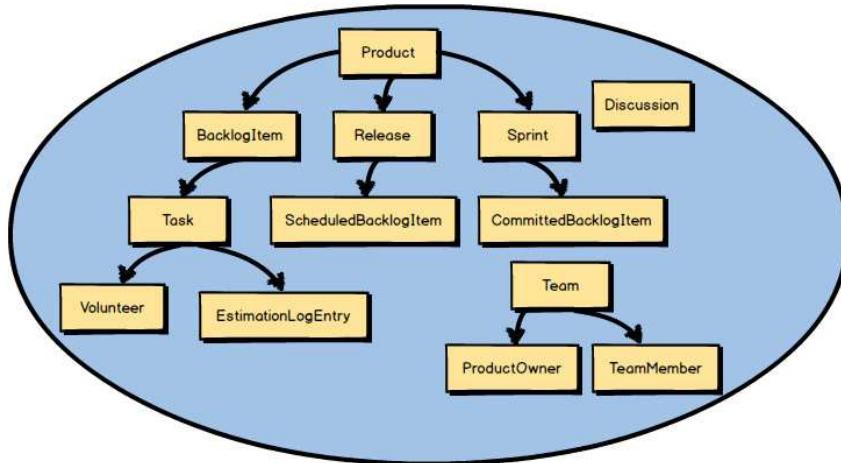
Entonces, se debe asumir la misión de luchar contra los *modelos de dominio anémicos* [IDDD]. Si se exponen métodos de *setters* públicos, eso podría conducir rápidamente a anemia, ya que la lógica para establecer valores en *Product* se implementaría fuera del modelo. Hay que pensar muy bien antes de hacerlo y tener en cuenta la advertencia.



```
public class Product : Entity
{
    ...
    public void PlannedProductBacklogItem(...)
    {
        ...
    }
}
```

Finalmente, se agrega un comportamiento complejo. Aquí aparecen cuatro métodos nuevos: *PlanBacklogItem()*, *PlannedProductBacklogItem()*,

`ScheduleRelease()` y `ScheduleSprint()`. El código C# para cada uno de estos métodos se debe agregar a la clase.



Recuerde que cuando se utiliza DDD, siempre se está modelando un *Ubiquitous Language* dentro de un *Bounded Context*. Por tanto, todas las partes del *agregado Product* se modelan según el *Ubiquitous Language*. Esas partes compuestas no se “inventan”. Todo muestra armonía entre el equipo unido de *expertos del dominio* y desarrolladores

## Elegir las abstracciones con sumo cuidado

Un modelo de software efectivo siempre se basa en un conjunto de abstracciones que abordan la forma de hacer las cosas de una empresa. Existe, sin embargo, la necesidad de elegir el nivel de abstracción apropiado para cada concepto que se está modelando.

Si se sigue la dirección del *Ubiquitous Language*, generalmente se crearán abstracciones adecuadas. Es más fácil modelar las abstracciones correctamente porque son los *expertos del dominio* quienes transmiten, al menos, la génesis del lenguaje de modelado. Sin embargo, a veces, existen desarrolladores de software con mucho entusiasmo para resolver problemas equivocados, que como resultado, intentarán forzar abstracciones extremas.

Por ejemplo, en el Contexto de Agile *Project Management* se trata

con Scrum. Entonces, tiene sentido modelar los conceptos de Product, BacklogItem, Release y Sprint que se han venido analizando. Sin embargo, ¿qué pasaría si los desarrolladores de software estuvieran menos preocupados por modelar el *Ubiquitous Language* de Scrum, y más interesados en modelar una solución para todos los conceptos actuales y futuros de Scrum?

Si se siguiera ese ángulo, los desarrolladores probablemente propondrían abstracciones como ScrumElement y ScrumElementContainer. Un ScrumElement podría cubrir la necesidad actual de Product y Backlog-Item, y ScrumElementContainer podría representar los conceptos obviamente más explícitos de Release y Sprint. El ScrumElement tendría una propiedad typeName, y se establecería en “Product” o “BacklogItem” en los casos apropiados. Así, se podría diseñar el mismo tipo de propiedad typeName para ScrumElementContainer y permitir que se establecieran los valores “Release” o “Sprint”.

¿Se perciben los problemas con este enfoque? Hay varios, pero tener en cuenta los siguientes:

- El lenguaje del modelo de software no coincide con el modelo mental de los *expertos del dominio*.
- El nivel de abstracción es demasiado alto, y habrá problemas graves cuando comience a modelar los detalles de cada uno de los tipos individuales.
- Esto conducirá a la creación de casos especiales en cada una de las clases y es probable que produzca una jerarquía de clases compleja con enfoques generales para problemas explícitos.
- Habrá mucho más código del necesario, porque se intenta resolver un problema sin solución que no debe importar en primer lugar.
- A menudo, el lenguaje de las abstracciones incorrectas se abrirá camino incluso en la interfaz de usuario, y causará confusión para los usuarios.
- Se perderá considerable tiempo y dinero.
- Nunca se podrá abordar previamente todas las necesidades futuras. En otras palabras, si se agregan nuevos conceptos Scrum en el futuro, el modelo existente fracasará en su previsión.

Seguir este camino puede parecer extraño, aunque a menudo se usa este nivel incorrecto de abstracciones en implementaciones de inspiración técnica.

Resulta importante no dejarse engañar por esta abstracta y atractiva trampa de implementación. Modelar el *Ubiquitous Language*, explícitamente en concordancia con el modelo mental de los *expertos en dominios*, refinado por el equipo. Al modelar las necesidades actuales del negocio, se ahorrará cantidad considerable en tiempo, presupuesto, código y momentos de desconcierto. Además, se hará un gran servicio a la empresa al modelar un *Bounded Context* preciso y útil, de tal manera que refleje un diseño efectivo.

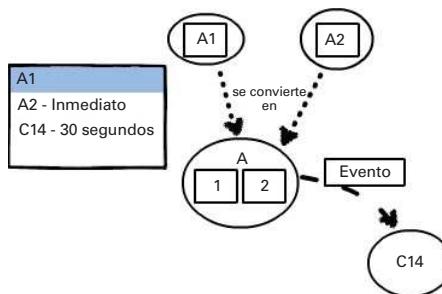
## Dimensionar agregados correctamente

Es muy probable que surja el interrogante de cómo determinar los límites de los *agregados* y evitar el diseño de grandes agrupaciones, mientras se mantienen los límites de consistencia que protegerán las verdaderas invariantes del negocio. Para eso, a continuación se proporcionará un enfoque de diseño útil. Si ya se han creado grandes *agregados*-clúster, es posible usar este enfoque para re-factorizarlos en agregados más pequeños; sin embargo, no se comenzará desde esa perspectiva. Considerar estos pasos de diseño ayudarán a alcanzar los objetivos de límites de coherencia:

1. Inicialmente, enfocarse en la segunda regla del diseño *agregado*, “*Designar agregados simples*”. Comienza por crear cada *agregado* con una sola entidad, que servirá como la *raíz del agregado*. Ni siquiera pensar en colocar dos *entidades* en un solo límite de momento, hasta que llegue el momento apropiado. Rellenar cada una de las *entidades* con campos/atributos/propiedades que se consideren están más estrechamente asociados a la única *entidad raíz*. Una clave importante en esta instancia es definir cada campo/atributo/propiedad que se requiera para identificar y hallar el *agregado*, así como cualquier campo/atributo/propiedad intrínseca adicional que se requiera para que el *agregado* se construya y se deje en un estado inicial válido.
2. A continuación se concentra la atención en la primera regla del diseño agregado, “*Proteger las invariantes del negocio dentro de los*

límites agregados”. Con el paso anterior, se asegura que, como mínimo, todos los campos/atributos intrínsecos estén actualizados cuando persiste el *agregado* de una sola entidad. Ahora es necesario examinar cada uno de los agregados en forma individual. A medida que se haga para el *agregado* A1, se pregunta a los *expertos del dominio* si cualquier otro *agregado* definido debe actualizarse en respuesta a los cambios realizados en el *agregado* A1. Luego, hacer una lista de cada uno de los agregados y sus reglas de consistencia, que indicarán los marcos de tiempo para todas las actualizaciones *basadas en reacción* (reaction-based updates). En otras palabras, “*agregado* A1” encabezaría la lista, y otros tipos de *agregados* estarían relacionados bajo A1 si se actualizaran como reacción a las actualizaciones A1.

3. Ahora se pregunta a los *expertos del dominio* cuánto tiempo puede transcurrir hasta que cada una de las *actualizaciones basadas en reacción* puedan tener lugar. Esto podrá conducir a dos tipos de especificaciones: (a) inmediatamente, y (b) dentro de N segundos / minutos / horas / días. Una forma posible de encontrar el umbral comercial correcto es presentar un marco de tiempo exagerado (semanas o meses, por ejemplo) que sea obviamente inaceptable. Esto probablemente hará que los expertos del negocio respondan con un marco de tiempo aceptable.
4. Para cada uno de los marcos de tiempo inmediatos (3a), hay que considerar la posibilidad de componer esas dos *entidades* dentro del mismo límite de *agregado*. Eso significa, por ejemplo, que el *agregado* A1 y el *agregado* A2 en realidad se compondrán en un nuevo *agregado* A [1,2]. A partir de ese momento, los *agregados* A1 y A2 ya no existirán como se definieron previamente, sólo existirá el *agregado* A [1,2].
5. Para cada uno de los *agregados* que se actualizan después de transcurrir determinado tiempo(3b), se actualizarán teniendo en cuenta la cuarta regla del diseño de agregados, “Actualizar otros agregados usando consistencia eventual”.



En esta figura el énfasis se encuentra en modelar el *agregado* A1. Obsérvese que en la lista A1 de reglas de consistencia, A2 tiene un marco de tiempo inmediato, mientras que C14 tiene un marco de tiempo eventual (30 segundos). Como resultado, A1 y A2 se modelan en un solo *agregado* A [1,2]. Durante la ejecución, el *agregado* A [1,2] publica un *Domain Event* que causa que el *agregado* C14 se actualice a la postre.

Igualmente, hay que tener cuidado con la posibilidad que el negocio insista en que todos los agregados se encuentren dentro de la 3a especificación (consistencia inmediata). Puede ser tendencia especialmente fuerte cuando muchos, en la sesión de diseño, están influenciados por el diseño de bases de datos y el modelado de datos. Estos tendrán un punto de vista muy concentrado en la transacción. Es muy poco probable, sin embargo, que el negocio realmente necesite una consistencia inmediata en todos los casos. Para cambiar esta forma de pensar, es probable que haya que emplear tiempo demostrando cómo las transacciones fallarían debido a actualizaciones simultáneas de múltiples usuarios en diferentes partes de esos grandes *agregados-clúster*. Además, se puede indicar cuánta sobrecarga de memoria habría con diseños de clúster tan grande. Obviamente, este tipo de problemas son exactamente lo que se intenta evitar.

Este ejercicio indica que la consistencia eventual es impulsada por el negocio, no por el equipo técnico. Por supuesto, habrá que encontrar una manera técnica de realizar actualizaciones entre múltiples *agregados*, tal como se explicó en el capítulo anterior sobre *Context Mapping*. Aun así, es únicamente el negocio el que puede determinar el marco de tiempo aceptable para que se realicen las actualizaciones entre varias *entidades*. Algunas son inmediatas, o transaccionales, lo que significa que deben ser administradas por el mismo *agregado*. Algunas son eventuales, lo que significa que pueden administrarse por ejemplo mediante

*Domain Events* y mensajes. También, resulta importante considerar lo que tendría que hacer la empresa si tuviera que ejecutar sus operaciones solo mediante sistemas de papel puede proporcionar información valiosa sobre cómo deben funcionar las distintas operaciones impulsadas por el dominio en un modelo de software de las operaciones comerciales.

## Unidades verificables

Asimismo, se deben diseñar *agregados* para que constituyan una encapsulación sólida para pruebas unitarias (*unit testing*). Los *agregados* complejos son difíciles de verificar. Seguir la guía de diseño anterior ayudará a modelar *agregados* verificables.

Una prueba unitaria es diferente del proceso de validación de las especificaciones de negocio (pruebas de aceptación), tal como se explica en el capítulo 2, “Diseño estratégico con *Bounded Contexts* y *Ubiquitous Language*”, y en el capítulo 7, “Herramientas de gestión y aceleración”. El desarrollo de pruebas unitarias se hará después de la creación de las pruebas de aceptación de especificación del escenario. Lo que concierne en pruebas unitarias es probar que el *agregado* hace correctamente lo que se supone que debe hacer. Conviene entonces comprobar todas las operaciones para garantizar la corrección, calidad y estabilidad de los *agregados*; así, se puede usar un marco de pruebas unitarias, y para ello hay literatura disponible sobre cómo realizar una prueba unitaria de manera efectiva. Esta prueba unitaria se asociará directamente al *Bounded Context* y se mantendrá con el repositorio de código fuente.

## Resumen

---

En síntesis, lo aprendido:

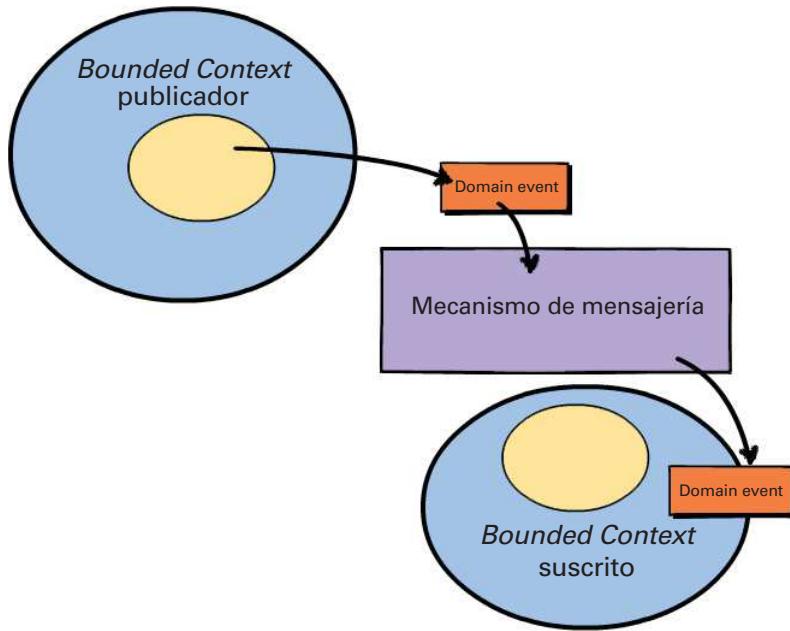
- Qué es el patrón *agregado* y por qué usarlo.
- La importancia de diseñar con un límite de consistencia en mente.
- Las diferentes partes de un *agregado*.
- Las cuatro reglas básicas del diseño efectivo de *agregados*.
- Cómo modelar la identidad única de un *agregado*.

- La importancia de los atributos *agregados* y cómo evitar la creación de un *Anemic Domain Model*.
- Cómo modelar el comportamiento en un agregado.
- Adherirse siempre al *Ubiquitous Language* dentro de un *Bounded Context*.
- La importancia de seleccionar el nivel de abstracción adecuado para los diseños.
- Una técnica para ajustar el tamaño correcto de las composiciones de *agregados*, y cómo eso incluye el diseño para la capacidad de pruebas.

Para un análisis más profundo de las *entidades*, *objetos de valor* y *agregados*, véanse los capítulos 5, 6 y 10 de *Implementing Domain-Driven Design* [IDDD].

# CAPÍTULO 6

## Diseño táctico con *Domain Events*



En capítulos anteriores se observó una introducción sobre cómo se usan los *Domain Events*. Un *Bounded Context* es el registro de un suceso relevante para el negocio en determinado *Bounded Context*. A estas alturas, se sabe que los *Domain Events* constituyen una herramienta muy importante para el diseño estratégico. Sin embargo, con frecuencia durante la fase de diseño táctico, los *Domain Events* se conceptualizan y pasan a formar parte del *Core Domain*.

Para comprender la importancia de un buen uso de los *Domain Events*, obsérvese el concepto de consistencia causal: un dominio empresarial proporciona coherencia causal si sus operaciones –que están relacionadas causalmente (una operación causa otra)–, se observan en ese mismo

orden por cada nodo dependiente de un sistema distribuido. Esto implica que las operaciones relacionadas causalmente ocurren en un orden específico, es decir, una *cosa* no puede suceder a menos que otra *cosa* suceda antes. Por ejemplo, un *agregado* no se puede crear o modificar hasta que se haya realizado una operación específica en otro *agregado*; por ejemplo:

1. Susana publica un mensaje que dice: “¡Perdí mi billetera!”
2. Gary responde, “¡Vaya, qué mala suerte!”
3. Susana publica luego un mensaje que dice: “No te preocupes, ¡ya encontré mi billetera!”
4. Gary responde: “¡Genial!”

Si estos mensajes se hubieran replicado sin un orden causal en nodos distribuidos, podría dar la impresión que “¡Genial!” se dice en respuesta a “¡Perdí mi billetera!”. Obviamente, el mensaje “¡Genial!” no está directa ni causalmente relacionado con “¡Perdí mi billetera！”, y no es como Gary quiere que lo entienda Susana, o cualquier otra persona que lo lea. Si la causalidad no se logra de la manera adecuada, el dominio general sería incorrecto, o como mínimo engañoso. Este tipo de arquitectura de sistema lineal y causal se puede lograr fácilmente mediante la creación y publicación de *Domain Events* en un orden correcto.

Gracias al diseño táctico, los *Domain Events* se convierten en una realidad en el modelo de dominio y, como resultado, se pueden publicar y consumir en el propio *Bounded Context* y por otros. En efecto, se trata de una forma poderosa de informar a los oyentes interesados sobre sucesos importantes. Ahora se podrá entonces aprender a modelar *Domain Events* y usarlos en *Bounded Contexts*.

## Diseñar, implementar y usar *Domain Events*

---

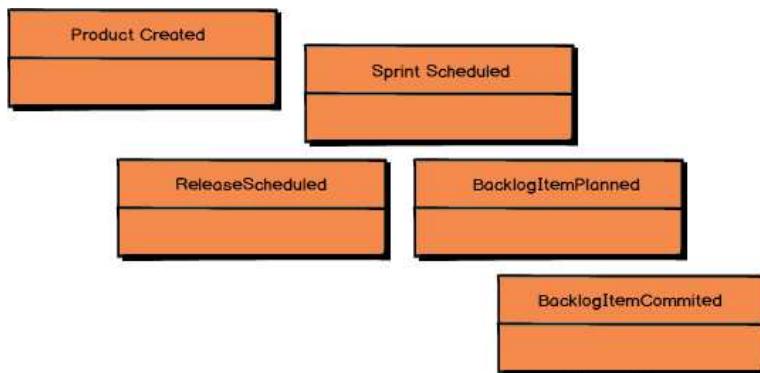
A continuación, una guía con los pasos necesarios para diseñar e implementar *Domain Events* en un *Bounded Context*. Más adelante, asimismo se podrán observar ejemplos de uso de *Domain Events*.



```
public interface DomainEvent
{
    public Date OccurredOn
    {
        get;
    }
}
```

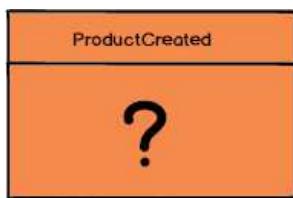
Este código en C# se podría considerar la interfaz mínima que todos los *Domain Events* deben implementar. Por lo general, se quiere transmitir la fecha y la hora en la que se produjo el *Bounded Context*, lo que es proporcionado por la propiedad `OccurredOn`. Esto no es absolutamente necesario, pero resulta bastante útil.

Probablemente serviría implementar esta interfaz en los *Domain Events*, los cuales se deben nombrar con sumo cuidado, pues las palabras deben reflejar el *Ubiquitous Language* del modelo. Estas palabras formarán un puente entre los sucesos del modelo creado y el mundo exterior. Por ello, es vital comunicar debidamente los sucesos.



Los nombres de los tipos de *Domain Events* deben constituir la declaración de algo ocurrido en el pasado, es decir, un verbo en tiempo pasado. Siguiendo con el contexto de *Agile Project Management*: `ProductCreated`, por ejemplo, indica que un Producto de Scrum se creó en algún momento en el pasado.

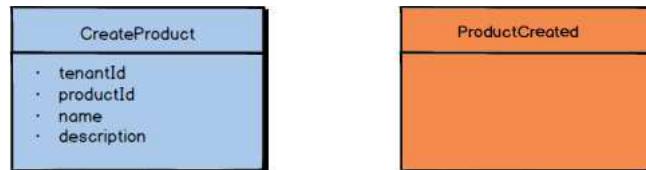
Otros ejemplos de *Domain Events* son `ReleaseScheduled`, `SprintScheduled`, `BacklogItemPlanned` y `BacklogItemCommitted`. Cada uno de los nombres indica de manera clara y concisa lo que sucedió en el *Core Domain*.



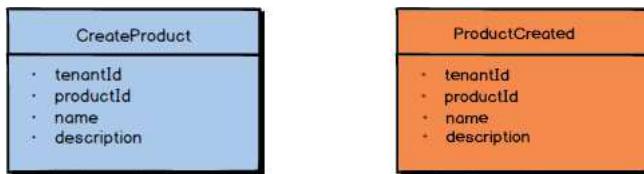
La combinación del nombre del *Bounded Context* y sus propiedades transmite completamente el registro de lo que sucedió en el modelo de dominio. Pero, ¿qué propiedades debe tener un *Bounded Context*?



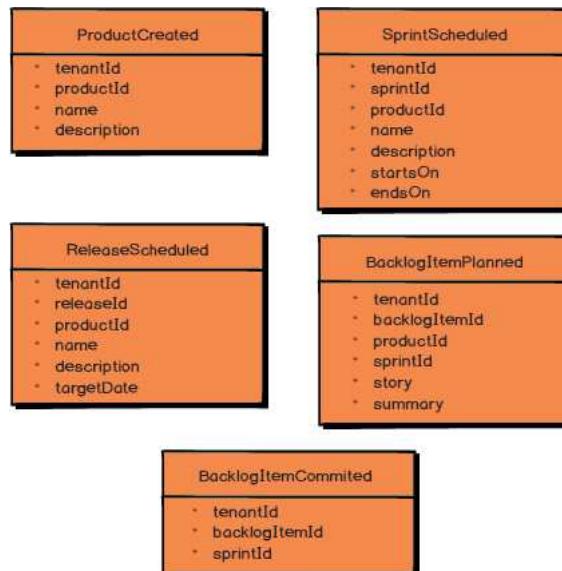
Aquí resulta importante preguntarse: “¿Qué es lo que estimula la aplicación para publicar un *Bounded Context*?”. En el caso de `ProductCreated` existe un comando que lo causa (un comando es solo un objeto que representa la solicitud de invocar un método o acción). Ese comando se llama `CreateProduct`. Se puede decir, entonces, que `ProductCreated` es el resultado del comando `CreateProduct`.



El comando `CreateProduct` tiene varias propiedades: (1) el `tenantId` que identifica el *tenant* suscriptor, (2) el `productId` que identifica únicamente el `Product` que se está creando, (3) `ProductName` y (4) `Product description`. Cada una de estas propiedades es esencial para crear un `Product`.

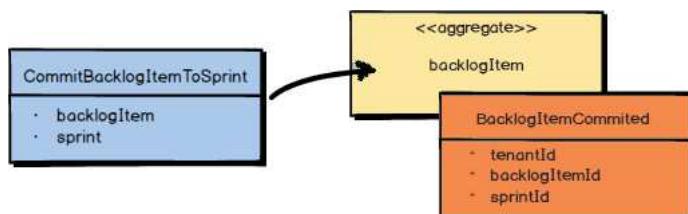


Por tanto, el `ProductCreated` del *Domain Event* debe contener todas las propiedades que se proporcionaron con el comando original que causó su creación: (1) `tenantId`, (2) `productId`, (3) `name` y (4) `description`. Esto informará de manera completa y precisa a todos los suscriptores sobre lo que sucedió en el modelo: se creó un `Product`, se identificó el *tenant* con el `tenantId`, el `Product` se identificó de forma única con `productId`, y el `Product` tenía el nombre y la descripción que se le asignó.

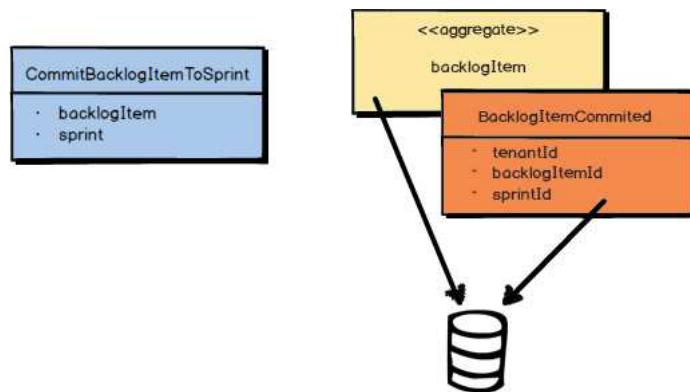


Estos cinco ejemplos dan una buena idea de las propiedades que deben incluirse con los varios *Domain Events* publicados por el *Agile Project Management Context*. Por ejemplo, cuando se comprometen *BacklogItem* a un *Sprint*, el *BacklogItemCommitted* del *Domain Event* se instancia y se publica. Este *Domain Event* contiene el *tenantId*, el *backlogItemId* del *BacklogItem* que se comprometió y el *sprintId* del *Sprint* al que se comprometió.

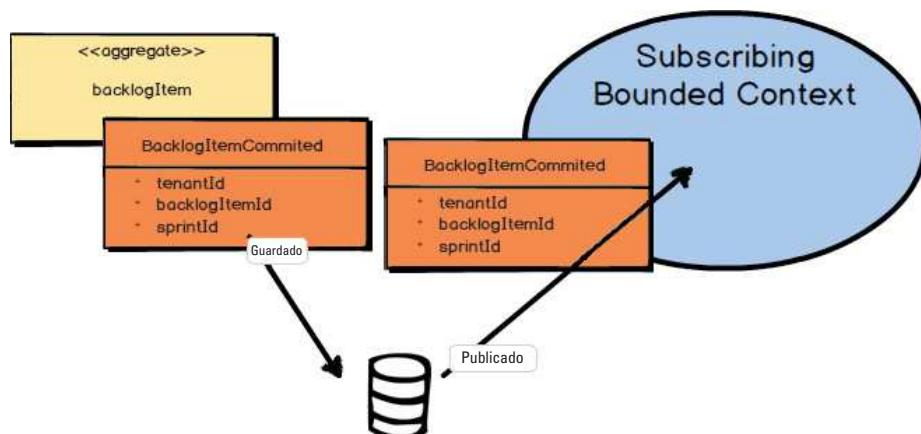
Tal como se describió en el capítulo 4, “Diseño estratégico con *Context Mapping*”, existen ocasiones en las que un *Domain Event* puede enriquecerse con datos adicionales. Esto es especialmente útil para consumidores que no quieren volver a consultar el *Bounded Context* para obtener datos adicionales. Evita, sin embargo, llenar un *Domain Event* con tantos datos que haga perder su significado. Por ejemplo, obsérvese el problema con *BacklogItemCommitted*, que mantiene todo el estado del *BacklogItem*. De acuerdo con este *Domain Event*, ¿qué sucedió realmente? Todos los datos adicionales pueden hacer que no quede claro y se requiera que el consumidor tenga conocimiento profundo del *BacklogItem*. Además, se considera usar *BacklogItemUpdated* con el estado completo de *BacklogItem*, en lugar de proporcionar *BacklogItemCommitted*. Los cambios en *BacklogItem* no quedan claros y el consumidor se ve forzado a comparar el último *BacklogItemUpdated* con el anterior *BacklogItem*.



Para comprender mejor el uso correcto de los *Domain Events*, obsérvese el siguiente escenario: el dueño del producto compromete un *BacklogItem* a un *Sprint*. El comando hace que se carguen el *BacklogItem* y el *Sprint*. Entonces, el comando se ejecuta en el agregado *BacklogItem*. Esto hace que el estado del *BacklogItem* se modifique y, a continuación, el *BacklogItemCommitted* del *Domain Event* se publica como un resultado.



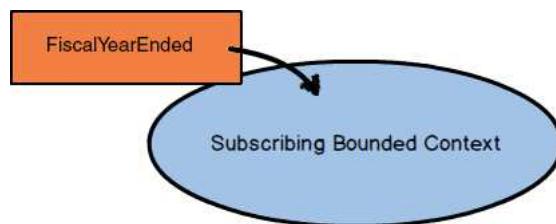
Es importante que el *agregado* modificado y el *Domain Event* se guarden de forma conjunta en la misma transacción. Si se utiliza una herramienta de mapeo relacional de objetos (ORM), se guarda el *agregado* en una tabla y el *Domain Event* en una tabla *Event Store* (almacenamiento de eventos) para después cerrar la transacción. Y si se utiliza *Event Sourcing*, el estado del agregado está completamente representado por los mismos *Domain Events*. En la siguiente sección de este capítulo se analizará la *Event Sourcing*. En cualquier caso, persistir el *Domain Event* en el almacén de eventos conserva su orden causal en relación con lo que ha ocurrido en el modelo de dominio.



Una vez que el *Domain Event* se guarda en el *event store*, se puede publicar a cualquiera de las partes interesadas, bien sea dentro del propio *Bounded Context* o a otros *Bounded Contexts* externos. De esta

manera, se anuncia al mundo que algo notable ha sucedido en el *Core Domain*.

Asimismo, tener en cuenta que guardar el *Domain Event* en su orden causal no garantiza que llegue a otros nodos distribuidos en ese mismo orden. Por tanto, es responsabilidad del *Bounded Context* consumidor reconocer la causalidad adecuada. Puede ser el tipo de *Domain Event* el que puede indicar la causalidad, o pueden ser metadatos asociados al *Domain Event*, como una secuencia o un identificador causal. La secuencia o el identificador causal indicarían qué causó este *Domain Event*, y si la causa aún no ha ocurrido, el consumidor debe esperar a que esa causa ocurra para aplicar el evento. En algunos casos, es posible ignorar *Domain Event* latentes cuando ya han sido sustituidos por las acciones asociadas a un evento posterior. En ese caso, la causalidad tiene un impacto irrelevante

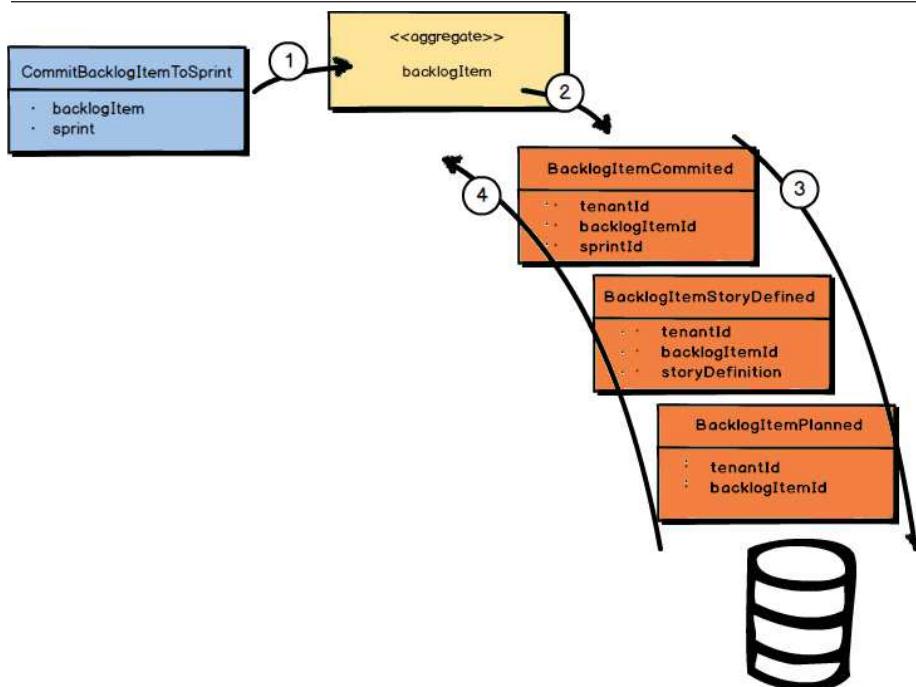


En este punto, vale la pena tratar un punto más acerca de qué puede causar un *Domain Event*. Normalmente, los *Domain Events* son causados por comandos de usuario emitidos por la interfaz de usuario. Puede haber, sin embargo, otras fuentes que los causen: por ejemplo, un temporizador que caduca al final de cada día hábil o al final de una semana, mes o año. En casos como este no será estrictamente un comando el que cause el evento, ya que la finalización de un periodo es un hecho en sí. No se puede rechazar que haya expirado un periodo, y si esto es relevante para el negocio, el vencimiento del tiempo se modela como un *Domain Event*, y no como un comando.

Asimismo, la expiración de un periodo por lo general tendrá un nombre descriptivo que se convertirá en parte del *Ubiquitous Language*. Por ejemplo, “fin de año fiscal” puede ser un evento relevante al que la empresa necesita reaccionar. Por ejemplo, en Wall Street la hora 4:00 p.m. (16:00) se referencia como “cierre de mercado” y no solo como 4:00 p.m. Por tanto, se tiene un nombre para ese *Domain Event* en particular, basado en un hito temporal.

Un comando, a diferencia de un *Domain Event*, puede ser rechazado como inadecuado dadas ciertas circunstancias, como la existencia y la disponibilidad de algunos recursos (producto, fondos, etc.) u otro tipo de validación de negocio. Por tanto, un comando puede ser rechazado, pero un *Domain Event* es un hecho y no se puede negar lógicamente. Aun así, en respuesta a un *Domain Event* basado en el tiempo, la aplicación podría generar uno o más comandos para pedirle a la aplicación realizar un conjunto de acciones.

## *Event Sourcing*



*Event Sourcing* se puede describir cuando persisten todos los *Domain Events* que han ocurrido para un *agregado* como registro histórico de lo que cambió en esa instancia del *agregado*. En lugar de persistir el estado final del *agregado* como un todo, se almacenan todos los *Domain Events* individuales que le han sucedido. Obsérvese cómo se puede hacer.

Todos los *Domain Events* que se han producido para un *agregado*, ordenados tal como se produjeron originalmente, conforman su flujo de eventos. El flujo de eventos comienza con el primer *Domain Event* que se produjo para el *agregado* y continúa hasta el último *Domain Event* sucedido. A medida que se producen nuevos *Domain Events* para determinado agregado, se agregan al final de su flujo de eventos. Volver a aplicar el flujo de eventos al *agregado* permite reconstituir su estado desde el sistema de persistencia de nuevo en la memoria. En otras palabras, cuando se utiliza *Event Sourcing*, un agregado que se elimina de la memoria se puede reconstituir completamente gracias a su flujo de eventos.

En el diagrama anterior, el primer *Domain Event* que ocurre es `BacklogItemPlanned`, el siguiente es `BacklogItemStoryDefined`, y el último en ocurrir es `BacklogItemCommitted`. El flujo de eventos está compuesto por esos tres eventos, y tal como se observa en la figura, siguen el orden descrito.

Cada uno de los *Domain Events* que ocurren para la instancia de un *agregado* son causados por un comando, tal y como se describió previamente. En el diagrama anterior, el comando `CommitBacklogItemToSprint` se acaba de usar y ha causado el `BacklogItemCommitted Domain Event` que ocurrirá.



El almacenamiento de eventos (*Event Store*) es solo una colección de almacenamiento secuencial o una tabla donde se anexan todos los *Domain Events*. Puesto que el almacenamiento de eventos solo se añade al final, el mecanismo de almacenamiento es extremadamente rápido, así que se puede esperar que un *Core Domain* que use *Event Sourcing* tenga un rendimiento muy alto, una latencia baja y una alta escalabilidad.

## Acerca del rendimiento

Si una de las prioridades es el rendimiento, es aconsejable saber un poco más sobre memoria caché y fotografías (*snapshots*). En primer lugar, los *agregados* de mayor rendimiento serán aquellos que se almacenen en la memoria caché, donde no será necesario reconstituirlos desde el almacenamiento a largo plazo cada vez que se utilicen. Usar el modelo Actor donde cada *agregado* es un Actor [Reactive] es una de las maneras más fáciles de mantener en la memoria caché el estado de los *agregados*.

Otra herramienta a disposición son las fotografías (*snapshots*), en los que el tiempo de carga de los *agregados* que han sido desalojados de la memoria puede reconstituirse de manera óptima sin recargar cada *Domain Event* de un flujo de eventos. En otras palabras, se podrá mantener una fotografía de un estado incremental del *agregado* (objeto, actor o registro) en la base de datos. En *Implementing Domain-Driven Design [IDDD]* y en *Reactive Messaging Patterns with the Actor Model [Reactive]*, se podrá analizar en detalle el tema de las fotografías (*snapshots*).

---

Una de las mayores ventajas de usar *Event Sourcing* es que se guarda un registro de todo lo que ha sucedido en el *Core Domain*, a nivel de ocurrencia individual. Esto puede ser muy útil para la empresa por muchas razones, algunas de ellas muy obvias en este momento, como el cumplimiento de ciertas legislaciones, el análisis de datos y otras que surgirán con el tiempo. Además de ventajas de negocios, tiene ventajas técnicas; por ejemplo, los desarrolladores pueden usar secuencias de eventos para examinar tendencias de uso y depurar su código fuente.

Mayor información sobre *Event Sourcing* se podrá encontrar en *Implementing Domain-Driven Design [IDDD]*. Usar *Event Sourcing* implica, en la mayoría de los casos, usar también CQRS, del cual también se encontrará más información en dicha obra.

## Resumen

---

En síntesis, lo aprendido:

- Cómo crear y nombrar *Domain Events*.
- La importancia de definir e implementar una interfaz estándar para los *Domain Events*.
- La importancia de nombrar apropiadamente los *Domain Events*.
- Definir las propiedades de los *Domain Events*.
- Cómo los *Domain Events* se pueden causar mediante comandos, o al detectar un cambio de estado como una fecha u hora.
- Cómo guardar los *Domain Events* en un almacenamiento de eventos.
- Cómo publicar *Domain Events* después de guardados.
- Qué es *Event Sourcing* y cómo los *Domain Events* se pueden almacenar y utilizar para representar el estado de los *agregados*.

Para mayor información sobre *Domain Events* e integración, véanse los capítulos 8 y 13 de *Implementing Domain-Driven Design* [IDDD].

## CAPÍTULO 7

# Herramientas de gestión y aceleración



Usar DDD es embarcarse en una aventura de aprendizaje profundo sobre el negocio, para luego modelar software en función de lo aprendido. Realmente se trata de un proceso de aprendizaje, experimentación, cuestionar conceptos, más aprendizaje y, finalmente, modelización. Se procesa y depura conocimiento en grandes cantidades para producir un diseño que sea efectivo y satisfaga las necesidades estratégicas de toda una organización. El desafío principal en las industrias aceleradas de hoy en día radica en el aprendizaje rápido. El tiempo y los plazos son, posiblemente, algunos de los factores que más influencian nuestras decisiones, quizás más de lo que deben: si no se entregan los productos a tiempo y con base en el presupuesto, independientemente de la solución software, se fracasará. Y toda la organización cuenta y depende del equipo para tener éxito a como dé lugar.

A veces, algunos se han empeñado en convencer a las directivas de que la mayoría de las estimaciones en proyectos no tienen valor ni utilidad. En realidad, no hay seguridad de los resultados que pueda producir ese argumento en general pero todos los clientes con los que se trabaja cotidianamente están sometidos a la presión de tener que entregar en ciertas fechas, lo que les obliga a limitar el tiempo que destinan al proceso de diseño e implementación. En el mejor de los casos, es una continua

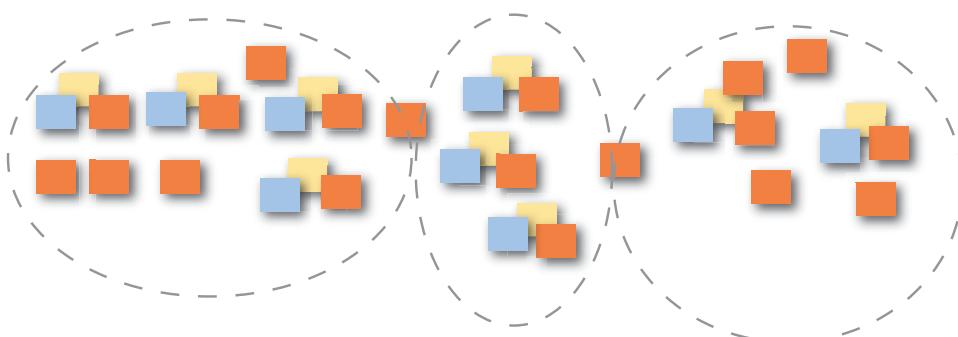
lucha entre el equipo encargado del desarrollo de software y la gerencia de la empresa.

Infortunadamente, una respuesta común a esta presión negativa es tratar de economizar y acortar los plazos eliminando la fase de diseño. Recuérdese lo que se ha venido trabajando desde el primer capítulo: el diseño es inevitable, y se fracasa con un diseño pobre o se triunfa con un diseño eficiente. Por tanto, la tarea es satisfacer las demandas de tiempo en forma estricta, diseñando de forma acelerada, utilizando enfoques que ayudarán a ofrecer el mejor diseño posible dentro de los límites de tiempo que hay que afrontar.

Para ello, en este capítulo, se trabajarán algunas herramientas muy útiles de gestión de proyectos y aceleración de diseño. Primero, se abordará la *Event Storming* y, a continuación, se intentará aprovechar los artefactos producidos por el proceso de colaboración para crear estimaciones significativas y, lo mejor de todo, que sean alcanzables.

## *Event Storming*

---



La *Event Storming* es una técnica de diseño rápido que pretende involucrar tanto a los *expertos del dominio* como a los desarrolladores en un proceso de aprendizaje acelerado. Se enfoca en el negocio y en sus procesos en lugar de hacerlo en sustantivos y datos.

Antes de trabajar la *Event Storming*, existía una técnica llamada *event-driven modeling*, que implicaba conversaciones, escenarios concretos y modelos centrados en eventos mediante un UML básico y sencillo. Los pasos específicos de UML se podían alcanzar con solo una pizarra,

aunque también se podían capturar en una herramienta. Sin embargo, como seguramente ya se sabe, pocas personas de negocios están informadas o son mínimamente hábiles en UML. Esas circunstancias significaban que el modelado recaía en un especialista u otro desarrollador que conociera los fundamentos de UML. Y de alguna manera, era un *modus operandi* útil, aunque hubiera la necesidad de involucrar más directamente a expertos del negocio en el proceso. Eso significaba, probablemente, dejar de lado UML en favor de otra herramienta más atractiva.

La técnica de *Event Storming* se dio a conocer hace varios años con Alberto Brandolini [Ziobrando], quien había experimentado también con otras formas de *event-driven modeling*. En una ocasión, por falta de tiempo, él decidió que debía deshacerse de UML y usar notas adhesivas. Este fue el origen de un enfoque de aprendizaje rápido y diseño de software que involucró a todos en la sala de manera muy directa en el proceso. Obsérvense algunas ventajas de este enfoque:

- Se trata de un enfoque directo y tangible. Todos reciben un bloc de notas adhesivas, un bolígrafo y son responsables de contribuir en las sesiones de aprendizaje y diseño. Tanto los gestores como los desarrolladores son iguales en el proceso y aprenden juntos. Todos aportan información al *Ubiquitous Language*.
- Todos se enfocan en los eventos y el proceso de negocio, y no en las clases y las bases de datos.
- Es un enfoque muy visual, que elimina el código de la experimentación y pone a todos en un mismo nivel, en el proceso de diseño.
- Es rápido y económico. Se puede elaborar un borrador de un nuevo *Core Domain* en cuestión de horas en lugar de semanas. Si uno escribe algo en una nota adhesiva y más tarde decide que no funciona, la descarta. Ese error cuesta uno o dos centavos, y nadie va a resistir la oportunidad de refinar gracias al esfuerzo invertido.
- El equipo alcanzará una comprensión mucho mayor del modelo de negocio. Es todo. Ocurre en cada ocasión. Siempre habrá los que vengan a la sesión pensando que ya tienen una buena comprensión del modelo de negocio principal, pero hasta aquellos tendrán mayor comprensión y nuevas perspectivas sobre el proceso empresarial.
- Todos aprenden algo. Bien sea un *experto del dominio* o un desarrollador de software, después de las sesiones habrá comprensión

clara del modelo en cuestión. Es importante hacer distinción entre esto y la comprensión del modelo de negocio: en muchos proyectos, algunos miembros del proyecto y posiblemente muchos, no entienden en qué están trabajando hasta que es demasiado tarde y el daño ya está en el código. Concebir un modelo ayuda a todos a aclarar malentendidos y avanzar en cierta dirección con propósitos unificados.

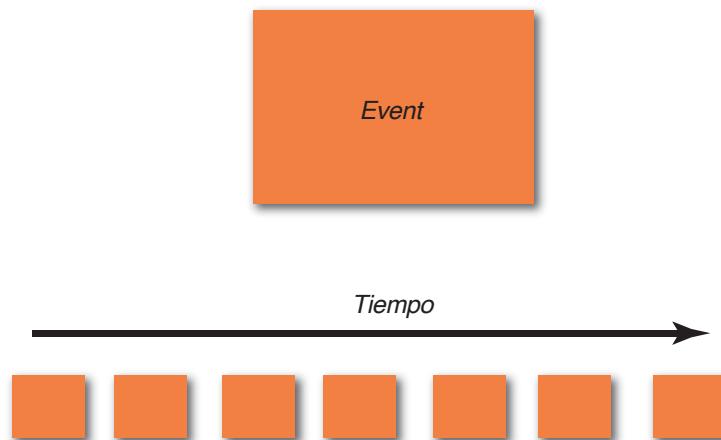
- Esto implica que también se identifican problemas en el modelo y en la comprensión de forma muy oportuna. Hay que solucionar los malentendidos y aprender de las dificultades. Todos en la organización se beneficiarán.
- Es posible usar la *Event Storming* para un modelado a muy alto nivel como para un modelado a nivel de diseño. En el modelado a alto nivel será menos preciso, mientras que una *Event Storming* a nivel de diseño llevará hacia artefactos específicos de software.
- No es necesario limitar la *Event Storming* a solo una sesión. Es posible comenzar con una sesión de dos horas y luego descansar. Es recomendable entonces respirar y dejar los avances por un momento para regresar al día siguiente y pasar una hora o dos más en profundizar y mejorar. Si se hace esto durante dos horas al día por tres o cuatro días, seguramente habrá comprensión profunda del *Core Domain* y las integraciones con los *subdominios* alrededor.

A continuación, una lista de personas, mentalidades y materiales necesarios para desarrollar un modelo con la *Event Storming*:

- Es esencial contar con las personas adecuadas, es decir, los *expertos del dominio* y los desarrolladores que trabajarán en el modelo. Todos tendrán preguntas y respuestas. Para poder darse apoyo mutuo, todos deben estar en la misma sala durante las sesiones de modelado.
- Todos deben acudir con mente abierta, libres de prejuicios. Un error grande es que durante las sesiones de *Event Storming* las personas intentan ser demasiado correctas y muy pronto. Por ello, hay que estar completamente dedicado a crear más eventos de los necesarios. Es mejor muchos eventos que pocos, porque eso será lo que hará aprender cada vez más. Habrá tiempo para refinamiento más tarde, y el refinamiento es rápido y económico.

- Importante asimismo tener a la mano un surtido de muchas notas adhesivas de varios colores. Como mínimo se necesitarán estos colores: naranja, púrpura/rojo, azul, amarillo, lila y rosa. Es posible que otros colores (como el verde; ver ejemplos posteriores) sean útiles. Las dimensiones de las notas adhesivas pueden ser cuadradas (3 pulgadas por 3 pulgadas o 7.62 cm por 7.62 cm) en lugar de la variedad más amplia que es rectangular. No se necesitará escribir mucho en la nota adhesiva, pocas palabras serán suficientes. Comprar notas adhesivas de pegamento extra fuerte, de tal modo que las notas no terminen en el suelo.
- Un rotulador negro para cada persona, lo que permitirá que la escritura a mano sea visiblemente clara en negrita. Los rotuladores de punta fina son los más aconsejables.
- Trabajar sobre una pared amplia donde se pueda modelar. El ancho es más importante que la altura, pero la superficie de modelado debe tener aproximadamente un metro/yarda de altura. El ancho debe ser últimamente ilimitado, pero como mínimo, 10 metros/yardas. En lugar de una pared de este tipo, siempre usar una mesa de conferencias o sobre el suelo. El problema con una mesa es que, en última instancia, se limitará el espacio del modelado. El problema sobre el suelo es que podría no ser accesible para todos en el equipo. La pared suele ser la mejor opción.
- Sobre un rollo largo de papel, como el que se puede encontrar en las tiendas de arte, de artículos escolares e incluso en tiendas populares de decoración. Debe tener las dimensiones descritas, con al menos 10 metros/yardas de ancho y 1 metro/yarda de altura. Se cuelga el papel en la pared con cinta adhesiva fuerte. Es posible no trabajar sobre papel y simplemente hacerlo en pizarras blancas. Esto puede funcionar temporalmente, pero las notas adhesivas tienden a perder adherencia en las pizarras, especialmente si se arrancan y se pegan varias veces. Los adhesivos se adhieren más tiempo cuando se aplican sobre papel. Si existe la intención de modelar durante tres o cuatro días, en lugar de hacerlo en una sola sesión, el tiempo de adherencia es importante.

Teniendo en cuenta el material básico y la participación de las personas adecuadas en la sesión, ya todos están listos para comenzar. Ténganse en cuenta cada uno de los siguientes pasos, uno por uno.



1. Desarrollar el proceso de negocio mediante una serie de Domain Events en notas adhesivas. El color más popular para usar en los Domain Events es el naranja: hace que los Domain Events se destaquen más en la superficie de modelado.

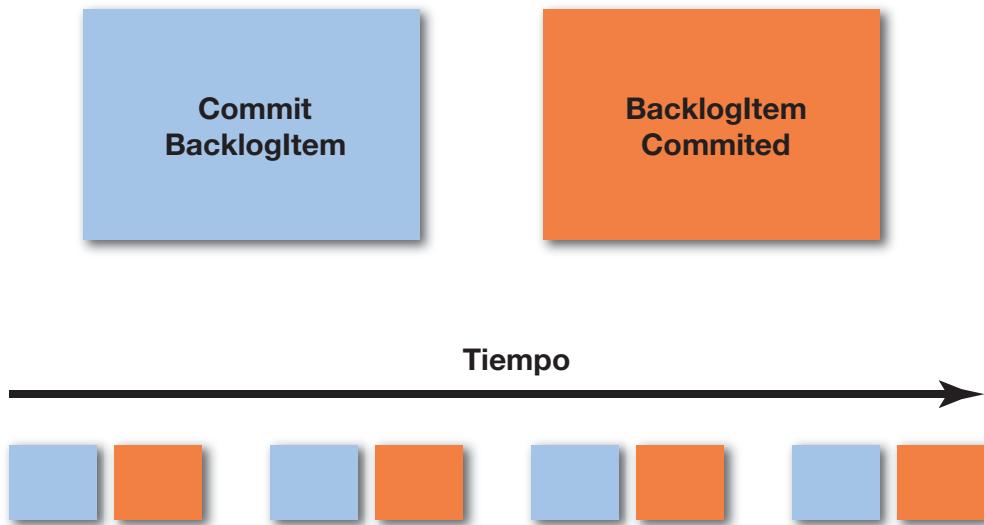
A continuación, algunas pautas básicas por aplicar mientras se crean los Domain Events:

- La creación de Domain Events como primer paso debe enfatizar en el primer y principal foco que es el proceso de negocio, no los datos o su estructura. El equipo puede tardar entre 10 y 15 minutos en prepararse para esto, pero seguir los pasos, tal y como se describen aquí. No tener la tentación de pasar más adelante.
- Escribir el nombre de cada Domain Event en una nota adhesiva. El nombre, como se aprendió en el capítulo anterior, debe ser un verbo en tiempo pasado. Por ejemplo, un evento puede llamarse `ProductCreated` y otro puede llamarse `BacklogItemCommitted`. Se pueden escribir estos nombres en varias líneas en las notas adhesivas. Si se hace una *Event Storming* estratégica y esos nombres son demasiado precisos para los participantes, usar otros nombres.
- Colocar las notas adhesivas en la superficie de modelado en orden cronológico, es decir, de izquierda a derecha en el orden como ocurre cada evento en el dominio. Comenzar con los primeros Domain Events en el extremo izquierdo y avanzar gradualmente hacia la derecha. Quizá algunas veces no habrá claridad sobre el orden cronológico, en cuyo caso simplemente colocar los Domain

*Events* en un lugar del modelo. Averiguar entonces la parte del “cuándo”, posiblemente se hará obvio más adelante.

- Un *Domain Event* que ocurre en paralelo a otro, siguiendo el proceso de negocio, puede ubicarse bajo el *Domain Event* que ocurre al mismo tiempo: utilizar el espacio vertical para representar procesamiento paralelo.
- A medida que se avanza en esta parte de la sesión del *Event Storming*, se encontrarán puntos problemáticos en un proceso de negocio nuevo o existente. Marcarlos evidentemente con una nota adhesiva púrpura/roja y un texto que explique por qué es un problema. Es necesario entonces invertir tiempo en esos puntos para aprender más sobre ellos.
- A veces, el resultado de un *Domain Event* es un *proceso* que debe ejecutarse. Podría representar un solo paso o varios pasos complejos. Cada *Domain Event* que causa la ejecución de un proceso debe capturarse y nombrarse en una nota adhesiva color lila. Trazar una línea con punta de flecha desde el *Domain Event* al *proceso* específico (nota adhesiva de color lila). Modelar un *Domain Event* en detalle únicamente si es importante para el *Core Domain*. Probablemente, el proceso de registro de un usuario sea necesario, pero no por ello se considera una característica principal de la aplicación. Modelar el proceso de *registro* como un solo evento general, *UserRegistered*, y continuar. Concentrar todos los esfuerzos en eventos más importantes.

Si, en determinado momento, se cree haber agotado todos los posibles *Domain Events* importantes, es posible que sea el momento de tomar un descanso y volver a la sesión de modelado más adelante. Sin duda, volver a la superficie de modelado un día después hará que se encuentren conceptos faltantes y afinar o eliminar conceptos superficiales que antes se habían considerado importantes. No obstante, llegará el momento en que se habrán identificado la mayoría de *Domain Event* de mayor importancia. Y es el momento de continuar con el siguiente paso.

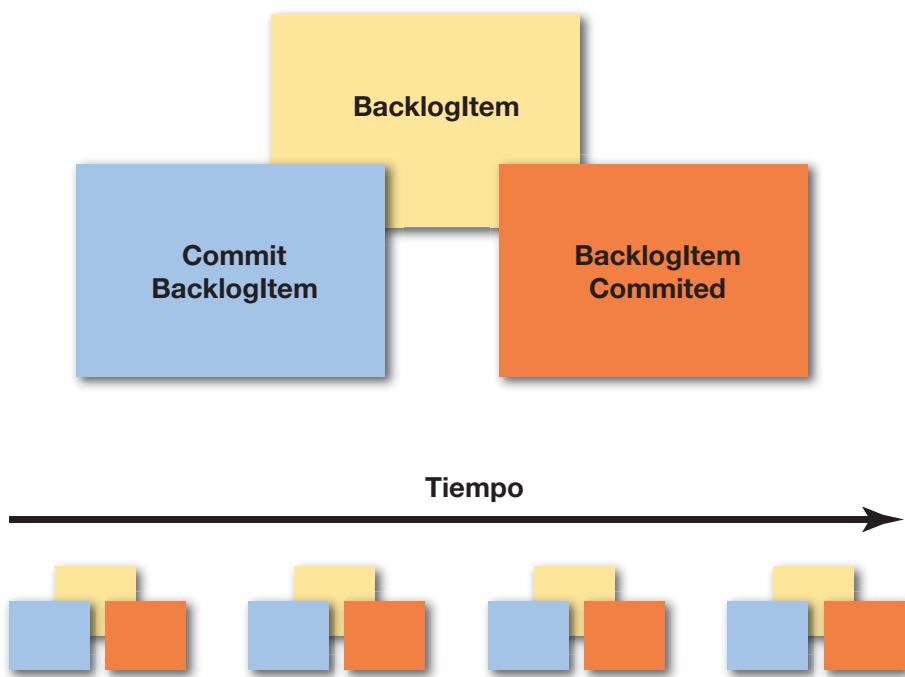


2. *Crear los comandos que causan cada Domain Event.* A veces, un *Domain Event* será el resultado de un suceso en otro sistema y, como resultado, fluirá en el sistema. Aun así, en la mayoría de los casos, un comando será el resultado de una acción del usuario, y causará un *Domain Event*. El *comando* debe indicarse en forma imperativa, como `CreateProduct` y `CommitBacklogItem`. A continuación, algunas pautas básicas:
  - En las notas adhesivas azules, escribir el nombre del *comando* que causa cada *Domain Event*. Por ejemplo, si se tiene un *Domain Event* llamado `BacklogItemCommitted`, el *comando* correspondiente que causa este evento se llamaría `CommitBacklogItem`.
  - Colocar la nota adhesiva azul del *comando* justo a la izquierda del *Domain Event* que causa. Se asocian en pares: *Comando/Evento*, *Comando/Evento*, *Comando/Evento*, etc. Recuérdese que algunos *Domain Events* ocurrirán debido a alcanzar ciertos límites de tiempo, y por tanto, es posible que no tengan un *Comando* correspondiente que los cause explícitamente.
  - Si existe un rol de usuario específico que realice una acción específica, y es importante determinarlo, es posible colocar una nota adhesiva amarilla brillante en la esquina inferior izquierda del co-

mando azul con el dibujo de un muñeco de palo y el nombre del rol. En la figura anterior, “dueño del producto” sería el rol que realiza el *comando*.

- A veces, un *comando* causará la ejecución de un *proceso*. Esto podría representar un solo paso o varios pasos complejos. Cada *comando* que cause la ejecución de un *proceso* debe capturarse y nombrarse en una nota adhesiva lila. Trazar una línea con una punta de flecha desde el *comando* hacia el *proceso* específico (nota adhesiva lila). Si el *proceso*, a su vez, causa uno o más *comandos* o *Domain Events*, y si ya se sabe cuáles son, crear notas adhesivas para ellos y representarlos de la misma forma.
- Continuar el movimiento de izquierda a derecha en orden cronológico, igual a cuando se traza por primera vez cada uno de los *Domain Events*.
- Es posible crear *comandos* que hagan pensar en *Domain Events* (tal como se ha mencionado anteriormente al descubrir los *procesos* lila u otros) que no se pudieron prever antes. Continuar y abordar este descubrimiento colocando el *Domain Event* recién descubierto en la superficie de modelado junto con su *comando* correspondiente.
- También es posible encontrarse con un solo *comando* causando múltiples *Domain Events*, lo que es normal. Modelar ese *comando* y colocarlo a la izquierda de los múltiples *Domain Events* que genera.

Una vez se tengan todos los comandos asociados a los *Domain Events* que causan, se podrá continuar con el siguiente paso.



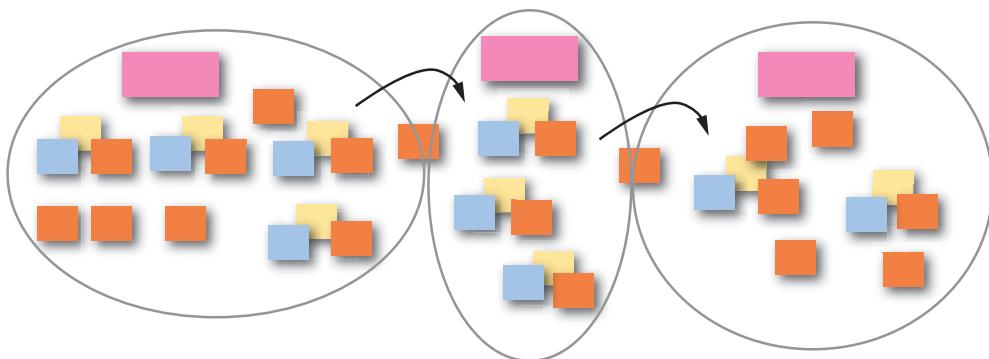
3. Asociar la entidad/agregado en el que se ejecuta el comando que produce además el Domain Event resultante. Este es el contenedor de datos donde se ejecutan *comandos* y se emiten *Domain Events*. Los diagramas de relaciones entre *entidades* son un primer paso muy popular en el actual mundo de TI; sin embargo, empezar por ellos es un gran error. La gente de negocios no los suele comprender bien, y empezar por ellos puede cerrar algunas conversaciones prematuramente. En la *Event Storming*, estos diagramas se relegan al tercer paso, ya que, como se dijo anteriormente, el enfoque se da más en el proceso de negocios que en los datos. En este punto, no obstante, una instancia que obliga a considerar los datos, i.e. en esta etapa, los expertos del negocio probablemente entenderán que los datos desempeñan un papel importante. Aquí algunas pautas para modelar los *agregados*:
  - Si a la gente de negocios no les gusta o les confunde la palabra *agregado*, usar otro nombre. Por lo general suelen comprender *Entidad*, o podrían llamarlo simplemente *Datos*. Lo importante es que la nota adhesiva permita al equipo clara comunicación so-

bre el concepto que representa. Usar las notas adhesivas amarillos para todos los *agregados* y escribir los nombres de cada agregado en una nota diferente. Los *agregados* se nombran con sustantivos, como `Product` o `BacklogItem`. Hacer esto para cada *agregado* en el modelo.

- Colocar la nota adhesiva del *agregado* detrás y ligeramente por encima de los pares *comando-Domain Event*. En otras palabras, poder leer el sustantivo escrito en la nota adhesiva del *agregado*, y los pares de *comando* y *Domain Event* se adhieren a la parte inferior para indicar que están asociados. Si se desea, poner un poco de espacio entre las notas adhesivas, pero tener claro cuáles pares *comando-Domain Event* pertenecen a cuál *agregado*.
- A medida que se avance en la línea cronológica del proceso de negocio, se darán cuenta de que usan los mismos *agregados* en varias ocasiones: no reorganizar la línea de tiempo para mover todos los pares de *comando/evento* bajo una sola nota adhesiva agregada. Más bien, crear el mismo agregado en varias notas adhesivas y colocarlas, en todos los pares *comando/evento* que le correspondan. La intención es modelar el proceso de negocio. Es más importante mantener la línea temporal.
- Es posible que al pensar en los datos asociados con varias acciones, se puedan descubrir nuevos *Domain Events*. No ignorar esto. Más bien, colocar los *Domain Events* recién descubiertos junto con los *comandos* y *agregados* correspondientes en la superficie de modelado. También se podría descubrir que algunos de los *agregados* son demasiado complejos, y será necesario dividirlos en un *proceso gestionado* (nota adhesiva lila). No hacer caso omiso a estas oportunidades.

Una vez completada esta parte de la etapa de diseño, algunos pasos opcionales. Si se usa la *Event Sourcing*, tal como se describió en el capítulo anterior, de hecho ya hay grandes avances hacia la comprensión de la implementación del *dominio principal*: *Event Storming* y la *Event Sourcing* que comparten varios rasgos comunes. Por supuesto, cuanto más cerca esté la *Event Storming* de la visión a alto nivel, más lejos estará de la implementación real. No obstante, se puede usar esa misma técnica para resaltar una vista a nivel de diseño. Por experiencia, los equipos tienden a fluctuar entre la estrategia de alto nivel y el nivel de diseño en una misma

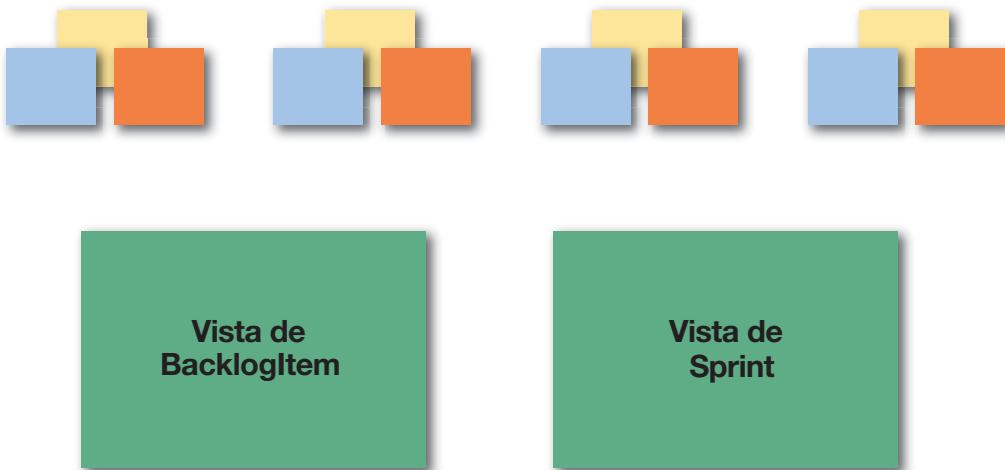
sesión. La necesidad de profundizar en ciertos detalles llevará más allá de la visión general, hacia un modelo de nivel de diseño allí donde sea necesario.



4. *Trazar límites y líneas con flechas para representar el flujo en la superficie del modelado.* Es muy probable que en las sesiones de *Event Storming* se descubra que hay múltiples modelos en juego, con *Domain Events* fluyendo entre sí. Obsérvese cómo lidiar con eso:
  - En resumen, es muy probable que haya límites en las siguientes circunstancias: divisiones departamentales, personas de negocios con definiciones en conflicto sobre el mismo concepto, o cuando un concepto es importante pero no forma parte del *Core Domain*.
  - Se pueden usar rotuladores negros para dibujar en el papel la superficie del modelado, don el propósito de mostrar el contexto y otros límites. *Utilizar líneas continuas para Bounded Contexts y líneas discontinuas para subdominios.* Obviamente, trazar estos límites en el papel será permanente, así que asegurarse de comprender este nivel de detalle antes de proceder. Si se desea, delimitar modelos de una forma menos permanente, usar notas adhesivas rosadas para demarcar áreas generales y esperar a marcar los límites con rotuladores permanentes hasta tener la confianza suficiente.
  - Poner las notas adhesivas rosadas dentro de varios límites y escribir el nombre que aplique dentro del límite en esas notas adhesivas. Esto servirá para nombrar los *Bounded Contexts*.

- Trazar líneas con puntas de flecha para mostrar la dirección de los *Domain Events* que fluyen entre *Bounded Contexts*. Esta es una forma fácil de representar cómo llegan algunos *Domain Events* al sistema sin ser causados por un comando en el *Bounded Context*.

Cualquier otro detalle sobre estos pasos debe ser obvio. Usar límites y líneas para comunicarse.



5. *Identificar las distintas vistas que los usuarios necesitarán para llevar a cabo sus acciones y roles importantes para diversos usuarios.*

- No están obligados a mostrar todas las vistas que proporcionará la interfaz de usuario; de hecho, no sería necesario mostrar ninguna de ellas. Si deciden mostrar algunas vistas, deben ser aquellas que sean significativas y requieran cuidado especial para crearlas. Estos artefactos de vista se pueden representar con notas adhesivas verdes en la superficie de modelado. Y si ayuda, dibujar una maqueta rápida (o estructura básica) de las vistas de la interfaz de usuario más importantes.
- También se pueden usar notas adhesivas amarillas para representar roles de usuario importantes. Mostrarlos solo si es necesario comunicar algo importante sobre la interacción del usuario con el sistema, o algo que haga el sistema para un rol específico.

Puede ser que opcionalmente haya que incorporar los pasos 4 y 5 a las sesiones de *Event Storming*.

## Otras herramientas

---

Por supuesto, esto no impide experimentar, por ejemplo, colocando otros bosquejos en la superficie del modelado o probando otros pasos de modelado en la sesión de *Event Storming*. Recuérdese que se quiere aprender y comunicar un diseño. Utilizar las herramientas que sean necesarias para modelar como equipo unido. Sólo tener cuidado de rechazar la ceremonia porque podría salir oneroso. A continuación, más ideas para ampliar las sesiones:

Introducir especificaciones ejecutables de alto nivel, que sigan el enfoque dado/cuando/entonces (given/when/then). Esto también se conoce como *pruebas de aceptación*. En el libro *Specification by Example* de Gojko Adzic [Specification], o en el capítulo 2, “Diseño estratégico con *Bounded Contexts* y el *Ubiquitous Language*”, se puede conocer un poco más sobre este punto, mediante la exposición de un ejemplo claro sobre este tipo de especificaciones. Tener cuidado de no exagerar con las *pruebas de aceptación* hasta el punto en que consuman todo y tengan prioridad sobre el modelo de dominio real. Por lo menos se requiere 15% a 25% más de tiempo y esfuerzo en usar y mantener especificaciones ejecutables que cuando se utilizan otros enfoques más comunes, basados en pruebas de unidad (también demostrados en el capítulo 2), y es fácil complicarse en el mantenimiento de especificaciones relevantes para la coyuntura de negocio, a medida que el modelo cambia con el tiempo.

Probar el mapeo de impacto (*Impact Mapping*) para asegurarse de que el software que se está diseñando es un *dominio principal*, no un modelo menos importante. Esta técnica también es definida por Gojko Adzic.

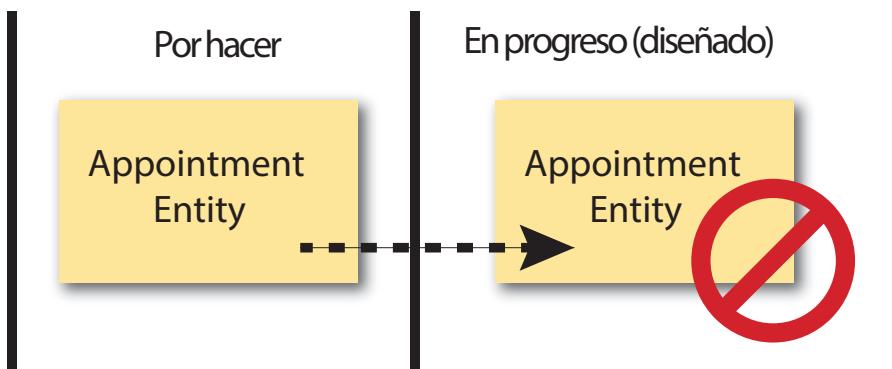
Consultar el mapeo de historias de usuario (*User Story Mapping*) de Jeff Patton, que se utiliza para dedicar esfuerzos al *dominio principal* y comprender en qué funciones del software se debe invertir.

Estas tres herramientas adicionales tienen gran coincidencia con la filosofía DDD y son muy adecuadas para cualquier proyecto DDD. Todas están creadas para usar en proyectos altamente acelerados. Además, tienen poca ceremonia y son muy económicas de usar.

## Cómo gestionar DDD en un proyecto Agile

Previamente se mencionó la existencia de un movimiento en torno a lo que se conoce como *no estimates* (no a las estimaciones). Ese enfoque rechaza los procesos comunes de estimación, tales como *story points* (puntos de historia) o *task hours* (estimar con horas). Este se concentra en dar valor al costo de control, y no estimar una tarea que probablemente requiera solo unos pocos meses para completar. No descartar este enfoque, pues aun así, hasta el momento, los clientes aún deben proporcionar estimaciones y colocar las tareas en marcos temporales, como esfuerzo de programación necesario para implementar incluso las funciones más detalladas. Si la opción *No Estimates* funciona en las circunstancias del proyecto, usarlo.

Asimismo, hay que ser consciente de que algunos miembros de la comunidad de DDD definen su propio proceso o su propio marco de ejecución para usar DDD con determinado proyecto. Esto funcionará de forma efectiva mientras sea aceptado por un determinado equipo, pero lo que puede resultar más difícil es que las organizaciones que ya hayan invertido Scrum compren un nuevo método.



Recientemente se ha observado que Scrum ha sido objeto de críticas considerables. Si bien es difícil tomar partido ante estas críticas, abiertamente hay que aceptar que a menudo Scrum se está utilizando de forma inapropiada. Ya anteriormente se mencionó la tendencia de muchos equipos a “diseñar” utilizando lo que se conoce como “la mecánica de las tareas”; en realidad, no es la forma como Scrum debe usarse en un

proyecto de software. Y, hay que repetirlo una vez más, la adquisición de conocimientos es tanto un principio de Scrum como un objetivo de DDD, pero se suele dejar en segundo plano a cambio de una entrega contundente y estricta, conseguida con Scrum. No obstante, Scrum sigue siendo un estándar para la industria y difícilmente será desplazado por otra metodología en un futuro cercano.

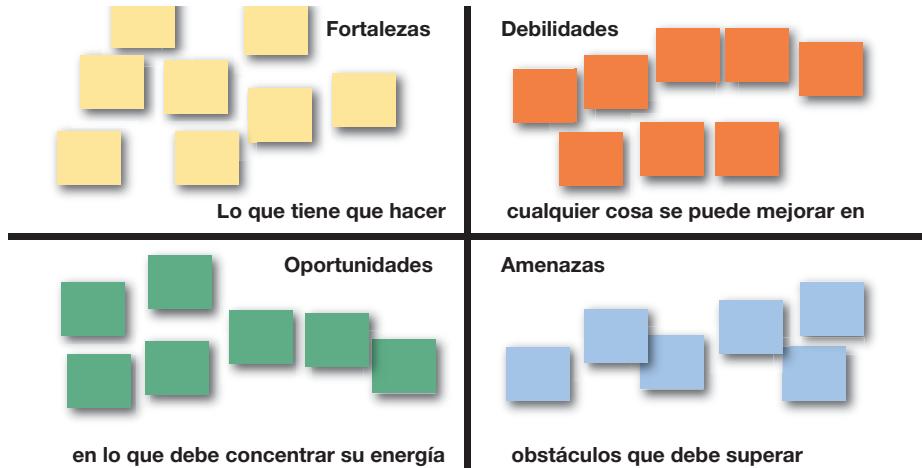
Por tanto, lo importante en esta instancia es mostrar cómo hacer que DDD funcione en un proyecto que usa Scrum. Las técnicas mostradas serán igualmente aplicables a otros enfoques de proyectos ágiles como Kanban. No habrá nada exclusivo de Scrum, aunque en algunas de las sugerencias se utilizará terminología propia de Scrum. Dado que muchas personas ya están familiarizadas con Scrum, al haberlo utilizado en algún momento, lo aconsejable es concentrarse en el modelo de dominio y aprendizaje, en la experimentación y el diseño con DDD. Si es preciso, entonces, buscar una guía general sobre el uso de Scrum, Kanban u otros enfoques ágiles.

Cuando se use el término *tarea* o *panel de tareas*, tener en cuenta que es compatible con métodos ágiles en general, incluso Kanban. Cuando se utilice el término *sprint*, también incluir las palabras reiteración para ágil en general y *work in progress*, WIP (trabajo en progreso) como referencia a Kanban. Puede que no siempre sea un ajuste perfecto, pero no se trata de definir un proceso real. La idea es beneficiarse de las ideas y encontrar una manera de aplicarlas adecuadamente en el *framework* de la ejecución ágil.

## Primero lo primero

Uno de los factores más importantes para triunfar con DDD en un proyecto es contratar buenos profesionales. Simplemente no hay remplazo para la gente técnicamente buena, en especial, desarrolladores. DDD es una técnica y filosofía avanzada para el desarrollo de software y requiere que los desarrolladores, incluso los muy buenos, la pongan en práctica. Nunca subestimar la importancia de contratar a las personas adecuadas con las habilidades adecuadas y automotivación.

## Utilizar análisis DOFA



Hay usuarios que quizás no estén familiarizados con el análisis DOFA (SWOT en inglés). La sigla traduce: debilidades, oportunidades, fortalezas y amenazas. Es una manera de pensar sobre el proyecto desde perspectivas muy específicas, pues se pretende tener máximo conocimiento de forma rápida. A continuación, las ideas básicas implícitas que conviene identificar en un proyecto:

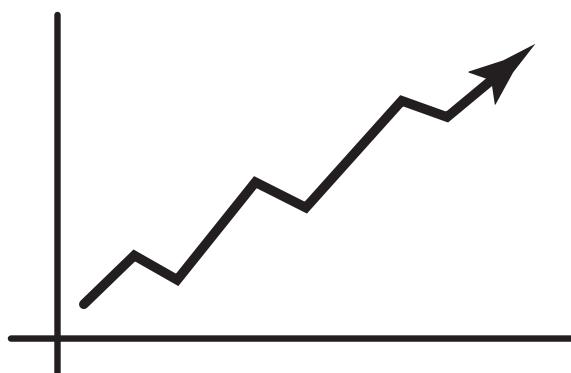
- *Debilidades*: características que ponen al negocio o al proyecto en desventaja con respecto a otros.
- *Oportunidades*: elementos que el proyecto podría aprovechar para su beneficio.
- *Fortalezas*: características del negocio o proyecto que le dan ventaja sobre los demás.
- *Amenazas*: elementos en el entorno que podrían causar problemas para la empresa o el proyecto.

En cualquier momento para cualquier proyecto que use Scrum u otra metodología ágil, hay que tomarse la libertad de usar el análisis DOFA con el propósito de determinar la situación actual del proyecto:

1. Dibujar una matriz grande con cuatro cuadrantes.
2. Al regresar a las notas adhesivas, escoger un color diferente para cada uno de los cuatro cuadrantes DOFA.
3. Luego, identificar las fortalezas, las debilidades, las oportunidades y las amenazas del proyecto.
4. Escribirlos en las notas adhesivas y colocarlos en la matriz en el cuadrante que corresponda. Utilizar el análisis DOFA (particularmente en el modelo de dominio) para planificar lo que se va a hacer al respecto. Los siguientes pasos para promover las áreas buenas y mitigar las áreas problemáticas serán decisivos para el éxito.

Aprovechar la oportunidad de colocar estas acciones en el tablero de tareas a medida que se realiza la planificación del proyecto, tal como se explica posteriormente.

### *Modeling Spikes y Modeling Debt*



No debe sorprender entonces que, en un proyecto de DDD, pueden surgir tareas inevitables por hacer como evolucionar el modelo o deuda en el modelo que pagar.

Entonces, una de las mejores decisiones que se pueden tomar en la fase de concepción de un proyecto es usar la *Event Storming*. Esta técnica y otros experimentos relacionados para mejorar el modelo forman lo que se conoce como *Modeling Spike* (tarea de investigación). Será necesario “comprar” conocimiento sobre el producto que se desarrolla

con Scrum, y algunas veces el pago es un *modeling spike*, y normalmente, durante la concepción de un proyecto investigar sobre el modelo es casi inevitable. Ya se ha hablado de cómo el uso la *Event Storming* puede reducir en gran medida el costo de la inversión necesaria.

No esperar que el modelo sea perfecto desde el principio, incluso si se cree que en la creación del proyecto ha habido o se planea tener investigación de gran valor. Ni siquiera será perfecto cuando se utilice la *Event Storming*. El negocio y la comprensión propia de él cambiarán con el paso del tiempo y así sucederá también con el modelo de dominio.

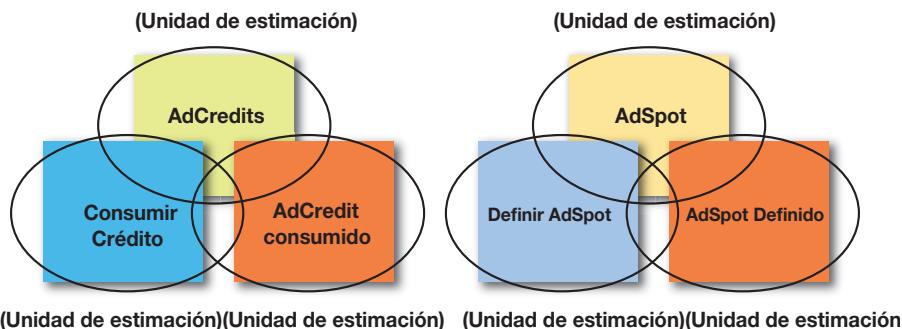
Además, si existe la intención de acotar en tiempo las tareas de investigación sobre el modelo y ponerlas en un tablero de tareas, hay que prever que se puede incurrir en deuda de modelo en cada sprint (o reiteración, o WIP). En otras palabras, no habrá tiempo para llevar a cabo todas las tareas de modelado deseadas a la perfección cuando hay limitación a un tiempo fijo. Por una razón, se comienza un diseño y, después de la experimentación, se darán cuenta de que no se ajusta a las necesidades del negocio tan bien como se esperaba. Sin embargo, el límite temporal marcado exigirá seguir adelante.

Lo peor que se podría hacer en ese momento, por consiguiente, es ignorar la necesidad de un diseño diferente y mejorado; sin embargo, no existe flexibilidad temporal para arreglarlo en ese momento. La solución en este punto es tomar nota de los cambios, y asignarlos a un *sprint* posterior (o reiteración, o WIP). Esto se puede llevar a una reunión retrospectiva<sup>1</sup> y se puede convertir en una nueva tarea en la próxima reunión de planificación de sprint (o reunión de planificación de reiteración, o se puede agregar a la cola de Kanban).

---

<sup>1</sup> N.T.: en Kanban, puede haber retrospectivas todos los días, así que no esperar mucho para exponer la necesidad de mejorar el modelo.

## Identificar tareas y estimar esfuerzos



La herramienta de *Event Storming* se puede utilizar en cualquier momento, no solo durante el inicio del proyecto. Durante cada sesión de *Event Storming*, orgánicamente se crearán varios artefactos. Cada uno de los *Domain Events*, *comandos* y *agregados* que se ubican en el modelo en el papel se pueden usar como unidades de estimación. Obsérvese cómo se hace.

Tipo de componente	Fácil (horas)	Moderada (horas)	Compleja (horas)
<i>Domain Event</i>	0.1	0.2	0.3
Comando	0.1	0.2	0.3
Agregado	1	2	3
...	...	...	...

Una de las maneras más fáciles y precisas de estimar es mediante un enfoque basado en métricas. Tal como se puede observar aquí, crear una tabla simple con las unidades de estimación para cada tipo de componente que se implementen. Esto eliminará las conjeturas sobre estimaciones y proporcionará conocimientos científicos sobre la estimación del esfuerzo. Así funciona la tabla:

1. Crear una columna para tipo de componente (*component type*), de tal modo que describa el tipo específico de componente para el cual se definen las unidades de estimación.
  2. Crear otras tres columnas, una para cada nivel de dificultad: fácil,

moderada y compleja. Estas columnas reflejarán la unidad de estimación, que se muestra en horas o fracciones de hora, para el tipo de unidad específica.

3. Ahora, crear una fila para cada tipo de componente en la arquitectura. En el ejemplo, se muestran los tipos de *Domain Events*, comando y agregado, pero no limitarse solo a esos. Crear una fila para los diversos componentes de la interfaz de usuario, servicios, persistencia, serializadores de *Domain Events* y deserializadores, y así sucesivamente. Tomarse la libertad de hacer una fila para cada tipo de artefacto que se creará en el código fuente. (Si, por ejemplo, normalmente se crea un serializador y un deserializador de *Domain Events* junto con cada *Domain Event* como un paso conjunto, asignar un valor de estimación a los *Domain Events*, de tal modo que reflejen la creación de todos estos componentes juntos en cada columna).
4. Luego, completar las horas o la fracción de hora necesaria para cada nivel de complejidad: fácil, moderado y complejo. Estas estimaciones incluyen el tiempo necesario para la implementación, además de todos los esfuerzos adicionales de diseño y prueba. Ser precisos y realistas.
5. Cuando se conozcan los ítems pendientes del Backlog (WIP) en las que se vaya a trabajar, contar con una métrica para cada una de las tareas e identificarlas con claridad. Usar una hoja de cálculo para esta tarea es lo más aconsejable.
6. Sumar todas las unidades de estimación para todos los componentes del *sprint* actual (reiteración o WIP), y esto se convertirá en la estimación total.

A medida que se ejecute cada sprint (reiteración o WIP), ajustar las métricas que reflejen las horas o fracciones de hora que realmente se requieran.

Aunque haya preferencia por usar Scrum y se deseche hacer estimaciones de horas, es aconsejable seguir estos pasos, pues es un método indulgente aunque también muy preciso: a medida que se coja el ritmo, se podrán ajustar las métricas de estimación para que ser más exactos y realistas. Podría incluso llevar algunos *sprints* hasta hacerlo bien, pero a medida que avance el tiempo y haya experiencia, se reducirán los tiempos y se podrá utilizar más las columnas fácil y moderado.

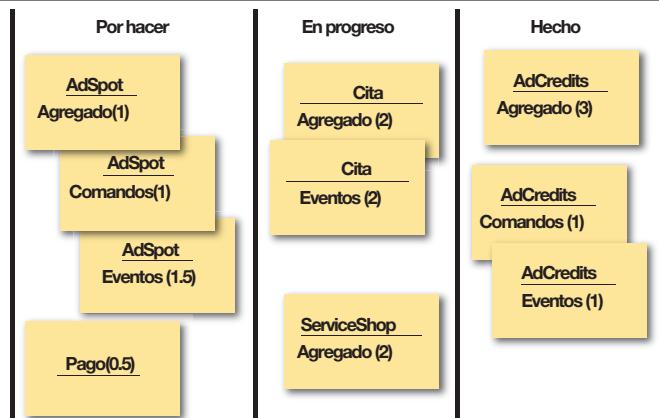
Si se utiliza Kanban y se considera que las estimaciones son mentira e innecesarias, preguntarse lo siguiente: ¿Cómo determinar un WIP preciso para delimitar correctamente la cola de trabajo? Independientemente de lo que se piense sobre las estimaciones, de cierta manera siempre se estará estimando el esfuerzo involucrado, y se espera que la estimación se cumpla. ¿Por qué no agregar un poco de ciencia al proceso y usar un enfoque de estimación sencillo y preciso?

### Sobre la precisión

En realidad, este enfoque funciona. En una ambiciosa iniciativa corporativa, una organización exigió estimaciones para un proyecto grande y complejo dentro de la propuesta general. Dos equipos fueron asignados a esta tarea. Primero, se involucró a un equipo de consultores de alto costo, acostumbrados a trabajar con compañías Fortune 500, para estimar y administrar los proyectos. Eran contadores, tenían varios doctorados y se equiparon con todo lo necesario para, a la vez, intimidar al equipo local y trabajar con clara ventaja. El segundo equipo, formado por arquitectos y desarrolladores, utilizó este enfoque de métricas para sus estimaciones. El proyecto estaba en el rango de los \$20 millones, y, al final, las estimaciones de los dos equipos solo variaron en \$200,000, el equipo técnico aportó la estimación más baja y eso no estuvo mal para unos técnicos.

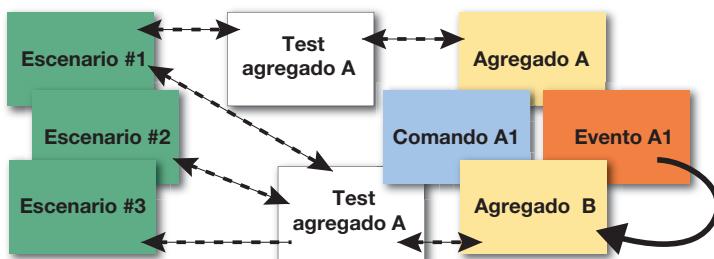
Por ello, debería haber precisión de 20% en estimaciones a largo plazo, y mucho mayor en estimaciones a corto plazo, para sprints, reiteraciones y colas WIP.

## Modelo acotado en tiempo (*Timeboxed Modeling*)



Ahora que se cuenta con las estimaciones para cada tipo de componente, las tareas se pueden basar directamente en esos componentes. Es posible mantener entonces cada componente como una sola tarea con una cantidad o fracción de horas, o se puede dividir las tareas un poco más. Es aconsejable tener cuidado al dividir las tareas; no acabar con tareas tan detalladas que configuren un panel de tareas demasiado complejo. Tal como se ha demostrado anteriormente, incluso podría ser mejor combinar todos los *comandos* y *Domain Events* utilizados por un único *agregado* en una sola tarea.

## Cómo implementar



Aunque se hayan identificado varios artefactos en la *Event Storming*, no necesariamente habrá todo el conocimiento necesario para trabajar en un escenario de dominio, historia y caso práctico específico. Si se requiere más, asegurarse de incluir en las estimaciones el tiempo para adquirir más conocimiento. ¿Tiempo para qué? Recuérdese que en el capítulo 2, se presentó la creación de escenarios concretos en torno al propio modelo de dominio. Esta puede ser una de las mejores maneras de tener conocimiento sobre el *Core Domain*, más allá de lo que se puede obtener de la *Event Storming*. Los escenarios concretos y la *Event Storming* son dos herramientas que deben usarse en conjunto. Observemos cómo funciona:

- Llevar a cabo una sesión rápida de *Event Storming*, alrededor de una hora. Seguramente se descubrirá que es necesario desarrollar escenarios más concretos en torno a algunos de los rápidos descubrimientos del modelo.
- Asociarse con un *experto del dominio* para analizar uno o más escenarios concretos que deben refinarse. Esto identifica cómo se utilizará el modelo de software. Recuérdese que estos procedimientos

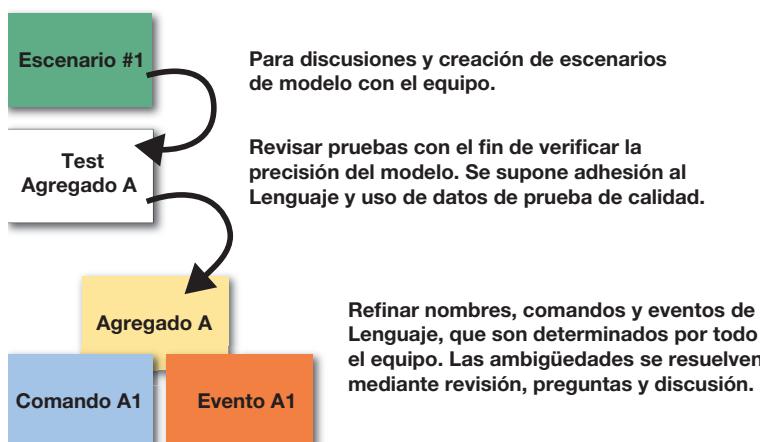
no son triviales, sino que deben hacerse con el objetivo de identificar elementos reales del modelo de dominio (por ejemplo, objetos), cómo colaboran y cómo interactúan con los usuarios (véase nuevamente el capítulo 2 si se quiere refrescar esta parte).

- Crear un conjunto de *pruebas de aceptación* (o especificaciones ejecutables) que ejerzan cada uno de los escenarios (véase el capítulo 2 según sea necesario).
- Crear los componentes para permitir que las pruebas/especificaciones se ejecuten. Repetir (en forma breve y rápida) a medida que se refinen las pruebas/especificaciones y los componentes hasta que hagan lo que espera el *experto del dominio*.
- Muy probablemente, parte de la reiteración (breve y rápida) haga considerar otros escenarios, crear pruebas/especificaciones adicionales, refinarse componentes existentes y crear otros nuevos.

Continuar con esto hasta que se hayan adquirido todos los conocimientos necesarios para cumplir con un objetivo de negocio determinado, o hasta que se agote el tiempo. Si no se alcanza la conclusión deseada, asegurarse de incurrir en deudas de modelado para que esto se pueda abordar en el futuro (cercano, preferiblemente).

No obstante, ¿cuánto tiempo serán necesarios los *expertos del dominio*?

## Cómo interactuar con los *expertos del dominio*



Uno de los principales desafíos de DDD es conseguir tiempo con los *expertos del dominio*, y solo usar el tiempo necesario. Muchas veces, los *expertos en el dominio* de un proyecto tienen muchas otras responsabilidades, reuniones y posiblemente viajes. Con tales ausencias potenciales del entorno del modelado, puede ser difícil contar con el tiempo suficiente con ellos. Por tanto, hacer que el tiempo con ellos sea significativo y limitarlo estrictamente a aquello que sea necesario. A menos que las sesiones de modelado sean divertidas y eficientes, se corre el riesgo de perder su ayuda en momentos decisivos. Si se logra que ellos tengan una imagen positiva de todo el proceso y del valor que este representa, es probable tener una buena sociedad, que seguramente será muy necesaria.

Las primeras preguntas por responder son: ¿Cuándo necesitamos tiempo con los *expertos del dominio*? ¿Para cuáles tareas es necesaria su ayuda?

- Incluir siempre *expertos del dominio* en actividades de *Event Storming*. Los desarrolladores tendrán muchas preguntas y los *expertos del dominio* tendrán sus correspondientes respuestas. Asegurarse de que estén juntos en las sesiones de *Event Storming*.
- En los análisis y en la creación de escenarios modelo será necesaria la información de los *expertos del dominio*. Retómese el capítulo 2 para los ejemplos.
- Se necesitarán *expertos del dominio* para revisar las pruebas que verifiquen el modelo. Esto supone que los desarrolladores ya han hecho un esfuerzo importante para adherirse al *Ubiquitous Language* y utilizar datos realistas y de calidad para las pruebas.
- Los *expertos del dominio* serán necesarios para refinar el *Ubiquitous Language* y los nombres de los *agregados*, *comandos* y *Domain Events* que fueron determinados por todo el equipo. Las ambigüedades se resuelven con base en revisiones, preguntas y discusiones. No obstante, las sesiones de *Event Storming* deben haber resuelto la mayoría de preguntas sobre el *Ubiquitous Language*.

Ahora que se sabe qué se necesita de los *expertos del dominio*, ¿cuánto tiempo requerir de ellos para cada una de las responsabilidades relacionadas anteriormente?

- La *Event Storming* debe limitarse a pocas horas (dos o tres) por cada sesión. Es posible que realizar sesiones en días consecutivos,

por ejemplo, durante tres o cuatro días.

- Reservar el tiempo que sea necesario para el análisis y el refinamiento del escenario, e intentar maximizar el tiempo para cada uno de estos, de tal modo que se pueda discutir e iterar de 10 a 20 minutos.
- Para las pruebas, será necesario tiempo adicional con los *expertos del dominio* para que revisen lo escrito; no esperar que se sienten a esperar mientras se escribe el código. Si lo hacen, mejor, aunque no esperar que así sea. Modelos precisos requieren menos tiempo para revisión y verificación. No subestimar la capacidad de los *expertos del dominio* para leer un test de aceptación sin ayuda. Pueden hacerlo, especialmente si los datos de prueba son realistas. Las pruebas deben permitir que el *experto del dominio* entienda y pueda verificar cada una en uno o dos minutos, aproximadamente.
- Durante las revisiones de pruebas, los *expertos del dominio* pueden proporcionar información sobre *agregados*, *comandos* y *Domain Events* y tal vez otros artefactos. Especialmente sobre cómo se adhieren al *Ubiquitous Language*. Esto se puede lograr sin gastar mucho tiempo.

Esta guía debe ayudar a usar la cantidad justa de tiempo con los *expertos del dominio* y limitar la cantidad de tiempo necesario de parte de ellos.

## Resumen

---

En síntesis, lo aprendido:

- La *Event Storming*, cómo se usa y cómo se pueden realizar sesiones con el equipo, con el fin de acelerar los esfuerzos de modelado.
- Sobre otras herramientas que se pueden usar junto con la *Event Storming*.
- Cómo usar DDD en un proyecto y cómo administrar las estimaciones y el tiempo con los *expertos del dominio*.

Para mayor información y más detallada sobre la implementación de DDD en proyectos, véase *Implementing Domain-Driven Design* [IDDD].

# Referencias

- [BDD] North, Dan. “Behavior-Driven Development.” 2006. <http://dannorth.net/introducing-bdd/>.
- [Causal] Lloyd, Wyatt, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. “Don’t Settle for Eventual Consistency: Stronger Properties for Low-Latency Geo-replicated Storage.” <http://queue.acm.org/detail.cfm?id=2610533>.
- [DDD] Evans, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley, 2004.
- [Essential Scrum] Rubin, Kenneth S. Essential Scrum: A Practical Guide to the Most Popular Agile Process. Boston: Addison-Wesley, 2012.
- [IDDD] Vernon, Vaughn. Implementing Domain-Driven Design. Boston: Addison-Wesley, 2013.
- [Impact Mapping] Adzic, Gojko. Impact Mapping: Making a Big Impact with Software Products and Projects. Provoking Thoughts, 2012.
- [Microservices] Newman, Sam. Building Microservices. Sebastopol, CA: O’Reilly Media, 2015.
- [Reactive] Vernon, Vaughn. Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Boston: Addison-Wesley, 2015.
- [RiP] Webber, Jim, Savas Parastatidis, and Ian Robinson. REST in Practice: Hypermedia and Systems Architecture. Sebastopol, CA: O’Reilly Media, 2010.
- [Specification] Adzic, Gojko. Specification by Example: How Successful Teams Deliver the Right Software. Manning Publications, 2011.
- [SRP] Wikipedia. “Single Responsibility Principle.” [http://en.wikipedia.org/wiki/Single\\_responsibility\\_principle](http://en.wikipedia.org/wiki/Single_responsibility_principle).
- [SWOT] Wikipedia. “SWOT Analysis.” [https://en.wikipedia.org/wiki/SWOT\\_analysis](https://en.wikipedia.org/wiki/SWOT_analysis).

## Referencias

[User Story Mapping] Patton, Jeff. User Story Mapping: Discover the Whole Story, Build the Right Product. Sebastopol, CA: O'Reilly Media, 2014.

[WSJ] Andreessen, Marc. "Why Software Is Eating the World." Wall Street Journal, August 20, 2011.

[Ziobrando] Brandolini, Alberto. "Introducing EventStorming." [https://leanpub.com/introducing\\_eventstorming](https://leanpub.com/introducing_eventstorming).